MACHINE INTELLIGENCE 2

EDITED BY
ELLA DALE
AND
DONALD MICHIE
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

EDINBURGH at the University Press

PREFACE

The readers of this book will very probably be familiar with its predecessor, *Machine Intelligence, I,* published under the same imprint and edited by N. L. Collins and Donald Michie, and so they will be aware of the fascination and familiar with the wide extent of the field of enquiry being investigated here, difficult as it may be to define it precisely.

The objectives vary exceedingly. We may start with: how to make a machine which can do what we could do (so that we will be free to do something else); how to make a machine which can do what we could do if we were more accurate and patient (so that we can eliminate human error); how to make a machine which can do what we could do, but do it more quickly (os that we can learn the answer to the problem before the problem changes).

From this we advance to: how to make the machine (and ourselves) more efficient on the average by defining sub-goals, by using sub-optimal search procedures, by throwing away information totally, partially, or temporarily, for the sake of getting something done within an acceptable time.

Next we have: how can we assist the machine to recognise limited successes; how can we use the machine's current experience to suggest more appropriate sub-goals; how in fact can we and the machine best act together?

We are now confronted with the questions: how can we converse with the machine during the period of operation; what is a suitable language for such a conversation; what can the machine best do, what can we best do, and how should our respective rôles be determined?

Thus a study of 'machine intelligence' leads after yet one more remove to the ancient imperative, 'Know thyself', and the universal problem of co-existence.

1967 DAVID KENDALL

CONTENTS

INTRODUCTION

What is Machine Intelligence? Is it a theory, a discipline, an engineering objective? Or is it a pretentious name for the more peculiar parts of computer science?

In our view the matter is quite clear. Since the objective is to bring into existence an intelligent machine, those engaged in the attempt must look on themselves as engineers. The relationship of this prticular brand of software engineering to the disciplines of psychology, logic, mathematics and computer science is thus the same as that which bridge-building bears to the fields of geometry, mechanics, materials science and hydrodynamics. There is a simple test of success in bridge-building. Walk actoss it, and you know that the bridge exists. If you can drive a twenty-ton truck across, the odds are that it is also well built. The equivalent test of an intelligent machine is, as Turing pointed out long ago, the test of rational conversation. No contemporary computer system can yet stand this test, but the time is now fast approaching when the first intelligent dialogues between man and machine will commence.

Growing awareness of this finds a certain reflection in the pattern of this new volume. The previous year's themes are there—both formal theory of programs and programming, and new experimentation with deductive and inductive systems. But the emphasis in the last four chapters on vehicles of communication, is more prominent than before, and marks a significant trend.

Once again our thanks are due to Oliver and Boyd; also to the University of Edinburgh for rewarding the labours of this Second Machine Intelligence Workshop with a liberal issue of refreshments at half time.

ELLA DALE
DONALD MICHIE

ABSTRACT FOUNDATIONS

1

SEMANTICS OF ASSIGNMENT

R. M. BURSTALL
EXPERIMENTAL PROGRAMMING UNIT
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

FORMAL SEMANTICS AND THE LAMBDA-CALCULUS

The past two or three years have seen a growing interest in the formal semantics of programming languages. The aims of these semantic investigations are twofold. First they give a means of specifying programming languages more precisely, extending the syntactic precision which was first achieved in the Algol report to murkier regions which were only informally described in English by the Algol authors. Secondly they give us new tools for making statements *about* programs, particularly for proving that two dissimilar programs are in fact equivalent, a problem which tends to be obscured by the syntactic niceties of actual programming languages.

Like most terminological questions, the dividing line between syntax and semantics has been the subject of much arid debate. The sense in which the words are used in this paper is not mysterious. Suppose that the definition of a programming language 'XTRAN' is published in two parts, 'Part A: Syntax' and 'Part B: Semantics'. After reading Part A, the reader should be able to say, when he is presented with a piece of text, 'Yes, this is an XTRAN program' or 'No, it is not'. But the program, given data, would print some results. The reader should be able to predict what these results would be when, and only when, he has read 'Part B: Semantics'.

One of the important aspects of programming language semantics is that it tries to connect the very empirical subject of programming with the main body of mathematics, particularly with mathematical logic. The part of logic which is most closely related to programming is that dealing with functions

and computability. P. J. Landin (1964, 1965, 1966a and b) and C. Strachey (1965) have sought to explain programming languages in terms of Church's calculus of lambda conversion (Church 1941, Rosenbloom 1950) also using ideas from Curry's combinatory logic (Curry & Feys 1958). The relevance of the lambda-calculus to programming and the use of recursive function definitions in proving equivalences were first brought out by McCarthy (1962, 1963).

A function can be described in two ways: first by saying how it is built out of certain simpler functions by combining them in various ways, secondly by saying what values it would yield when applied to various arguments. The lambda-calculus gives an unambiguous notation for describing the way a function is constructed from simpler primitive functions, and gives conversion rules for converting a function into another which is differently constructed in terms of the primitives but which would nevertheless give the same result when applied to the same arguments. There are simple and fully formalised rules for evaluating expressions of the lambda-calculus; if we can translate the application of a program to its data into an expression of this calculus we have fully specified the outcome.

Landin's 1965 and 1966b papers and Strachey's paper already cited are devoted to the task of translating Algol and CPL respectively into the lambda-notation. The parts of programming languages furthest from the purely functional lambda-notation are the imperative features: assignment and jumps. Landin translated Algol into a version of the lambda-notation which was extended to include assignments; these assignments were defined in terms of a 'sharing machine'. Strachey showed how CPL (or Algol for that matter) could be translated into purely functional lambda-notation by introducing a 'store' with appropriate 'contents' and 'update' operations defined on it. Although these methods serve well as a formal definition of the languages concerned, they leave something to be desired as a means of simplifying programs and proving equivalences between them. Landin (1966) gives a number of equivalence rules but these do not apply to the assignment expressions in his system. Strachey's translations in terms of a store lead to elaborate and rather impenetrable expressions as the translations of quite straightforward piece of CPL text.

The first part of this paper gives methods of translating many instances of assignment in ordinary programming languages into purely functional notation, producing relatively simple and manipulable expressions as the result. In the second part a method similar to Strachey's is used to tackle the problems arising in procedures with non-local variables and in assignment to data-structures with shared components.

This second method of translation is more powerful and general, but it produces more complex translations. It uses the concepts of address and storage location, in an abstract form. For the simpler uses of assignment, such as assignment to a temporary variable to avoid repeated evaluation of a common sub-expression or repeated assignments to a variable in a loop,

the more general method is unnecessary; the same effect can easily be obtained by a functional expression not involving addresses or locations. But in more complex cases, such as certain forms of non-local variable, call by 'reference' ('simple name') and data structure manipulation, the ideas of addresses and locations cannot be dispensed with by a rearrangement of the program before it is run.

I use Algol as the programming language to be translated, because it is well known and well documented, and extend it in various ways to make points about features of other programming languages. I have talked above about Church's 'lambda-notation'. It is not really the notation which concerns us here, but the formal system of the lambda-calculus. In fact I will translate into a dressed up (syntactically sugared) notation for expressions of the lambda-calculus, due to Landin. This also includes conditional expressions and operations on lists, all of which may be reduced to pure lambda-notation if desired.

Although there are still a number of unresolved questions about the techniques used in this paper I hope that it may help to stimulate further discussion of formal systems for expressing the semantics of programming languages.

ISWIM—A MORE INTELLIGIBLE VERSION OF THE LAMBDA-NOTATION

Landin (1966a) has given a detailed formal definition of a functional programming language which he calls ISWIM (If you See What I Mean). This language apart from two features can be rewritten in the lambda-notation of Church in a quite straightforward way, using the relationship between definitions and the lambda-notation explained by Landin (1965, 1966a). The features which cannot easily be rewritten are 'program points' and 'assignment'. Program points are a means of translating labels and hence they are not relevant to the subject matter of this paper. My aim here will be to translate Algol, with some extensions, into ISWIM without assignment. This is equivalent to translating into the lambda notation but the translations will be more readable. As Landin (1966a) has pointed out, the purely functional ISWIM expressions which will be obtained are susceptible to various transformations which preserve equivalence. Thus the translation into functional ISWIM could provide a means for proving that two Algol programs are equivalent.

I hope that the following examples will make clear the meaning of expressions in this very simply structured language without repeating Landin's formal definition. In the interests of clarity I have taken one or two slight liberties with Landin's notation as follows:

(*a*) Shape of brackets. Landin uses only parentheses. I will use parentheses for arguments of functions, diamond brackets ('⟨' and '⟩') for explicitly written lists and square brackets to indicate the scope of operators. A vertical stroke will be used above and below square

brackets as a meaningless, purely decorative character to assist the eye.

(*b*) Landin uses 'h' and 't' for the list selector operations (LISP 'CAR' and 'CDR'); I will use 'head' and 'tail'.

(*c*) The meaningless, purely decorative word **'result'** will sometimes be used when an expression is preceded by a number of definitions, e.g., [**let** $x=9$; **let** $y=u+v$; **result** $\langle x, y, u\rangle$]

(*d*) I will abbreviate 'undefined' to 'undef'.

ISWIM expressions will be written on the left and their values on the right.

(i) Basic functions

I assume for purposes of illustration the arithmetic operators '+', '−' and '×' and the list-processing functions 'cons', 'head' and 'tail'. The empty list will be denoted by 'nil'.

(ii) Conditional expressions

These are written '$p\rightarrow x, y$' (cf. Algol '**if** p **then** x **else** y').

(*a*) $1<2\rightarrow 3, 4$	3
(*b*) $1>2\rightarrow 3, 4$	4

(iii) Definitions

Definitions are introduced by **'let'** and a sequence of auxiliary definitions may precede any expression.

(*a*) **let** $x=2$; $1+x\times x$	$1+2\times 2$
(*b*) **let** $x=1$ **and** $y=2$; $x+y$	$1+2$
(*c*) **let** $x=1$; **let** $y=x+2$; **let** $x=4$; **result** $x+y$	$4+3$

Note that the definitions may refer to variables previously defined. Remember that **'result'** is a decorative word which may be omitted.

(*d*) **let** $x=1$; **let** $x=2$ **and** $y=x$; **result** $\langle x, x+y, y\rangle$	$\langle 2, 2+1, 1\rangle$

Note that in the second line x and y are defined in parallel and y takes the previous value of x. The result is a list of 3 items.

'let $x=2$ **and** $y=x$' is also written **'let** $\langle x, y\rangle=\langle 2, x\rangle$'.

(*e*) **let** $f(x)=x=1\rightarrow 0, x+1$; $f(3)$	4
(*f*) **let** $f(y)=y\times y$; **let** $f(x)=x=1\rightarrow 1, x\times f(x-1)$; $f(4)$	$4\times 3\times 3$

Note the definition of f is *not* recursive. It uses the previous version of f.

(*g*) **let rec** $f(x)=x=1\rightarrow 1, x\times f(x-1)$; $f(4)$	$4\times 3\times 2\times 1$

Note: **'rec'** signifies that the f in the right of the definition is the same as that being defined.

(*h*) **let** $f(x)=$[**let** $g(y)=y\times y$; | **let** $z=x+1$; | $\langle z, g(g(z))\rangle$]
$f(3)$ $\langle 4, (4\times 4)\times(4\times 4)\rangle$

(*i*) **let** $x=0$ **and** $y=0$;
let $\langle x, y\rangle=$[**let** $x=x+1$; | **let** $y=y+2$; | **result** $\langle x, y\rangle$];
result $x+y$ $1+2$

It is important for what follows that the reader should grasp the sequence of definitions here. This expression could equally well be written:

let $x=0$ **and** $y=0$;
let $\langle u, v\rangle=$[**let** $p=x+2$; | **let** $q=y+2$; | **result** $\langle p, q\rangle$];
result $u+v$ $1+2$

(iv) Printing

The word 'print' followed by an expression causes it to be printed

(*a*) **print** [**let** $x=2$; $\langle x, x\times x\rangle$] prints 2, 4

TRANSLATION OF SIMPLE ASSIGNMENTS AND CONDITIONAL STATEMENTS

What does the following piece of Algol program do?

$x:=1;$
$y:=2;$
$x:=x+y;$
$y:=y+3;$

If we run it and watch what happens, the answer would be 'nothing', at least nothing is printed. More usefully we may say that two new values are computed, and x and y are defined to take these values in what follows. I suggest that we should *not* try to find an ISWIM expression as a translation or explanation of this piece of code. It corresponds rather to a fragment of an ISWIM expression, thus

let $x=1$;
let $y=2$;
let $x=x+y$;
let $y=y+3$;

This is a sequence of ISWIM definitions which could equally well have been written

let $u=1$;
let $v=2$;

```
let x=u+v;
let y=v+3;
```

The definitions only make sense if x and y are used in some following expression. Thus we have a correspondence between

Algol

```
begin integer x, y;
  x:=1;
  y:=2;
  x:=x+y;
  y:=y+3;
  outinteger(1, x×y)
end
```

ISWIM

```
print [let x=undef and y=undef;|
      |let x=1;                |
      |let y=2;                |
      |let x=x+y;              |
      |let y=y+3;              |
      |result x×y              ]
```

This transcription seems so simple that one's first reaction is that it must be wrong. However, it is possible to push this correspondence between assignments and definitions a surprisingly long way. To do this we need the concept of the 'assignable variables' of a piece of Algol program. We define these as a list of all the variables which are, or might be, assigned to. We may then say that a piece of Algol program produces a list of new values for its assignable variables.

This is immediately useful when we consider conditional statements.

Algol

```
if x>0 then
  begin x:=x+1;
        y:=x+y
  end
else
  begin x:=x+2;
        z:=x+1
  end;
outinteger(1, x+y+z)
```

ISWIM

```
print [let ⟨x, y, z⟩=[x>0→          |
                      |[let x=x+1;  ||
                      ||let y=x+y;  ||
                      ||⟨x, y, z⟩   ]|
                      |,            |
                      |[let x=x+2;  ||
                      ||let z=x+1;  ||
                      ||⟨x, y, z⟩   ]];
      x+y+z]
```

The conditional has $\langle x, y, z\rangle$ as assignable variables and it is translated into a definition of x, y and z which has, as right-hand side, a conditional expression denoting a triple. This may be clearer if we distinguish the different variables all called x above, by renaming them $x1$, $x2$, etc.

```
print [let ⟨x4, y4, z4⟩=[x1>0→          |
                         |[let x2=x1+1; ||
                         ||let y2=x2+y1;||
                         ||⟨x2, y2, z1⟩  ]|
                         |,            |
                         |[let x3=x1+2; ||
                         ||let z3=x3+1; ||
                         ||⟨x3, y1, z3⟩  ]];
      x4+y4+z4]
```

In general we may let X and Y be lists of variables and $X:::Y$ be the list formed by merging the elements of X and Y without repetitions. We assume variables are in some canonical order, e.g., alphabetical or order of declaration.

We let $S\{X\}$ be an Algol statement or sequence of statements with X as its list of assignable variables, and $D\{X\}$ be a sequence of ISWIM declarations which translate it. We use E for an Algol or ISWIM expression.

(i) If $S\{X\}$ is '$x1:=x2:= \ldots xk:=E$' where $\langle x1, \ldots, xk\rangle$ is X then $D\{X\}$ is '**let** $x1=E$ **and** $x2=E$ **and** . . . **and** $xk=E$'.

(We will discuss later the case where evaluating E produces side-effects.)

(ii) '$S1\{X\}$; $S2\{X\}$' corresponds to '$D1\{X\}$; $D2\{X\}$'

(iii) 'E **then** $S1\{X1\}$ **else** $S2\{X2\}$' corresponds to
let $X1:::X2=[E\rightarrow[D1\ \{X1\};\ X1:::X2],\ [D2\{X2\};\ X2:::X2]]$.

(iv) '**if** E **then** $S\{X\}$' corresponds to '**let** $X=[E\rightarrow[D\{X\};\ X],\ X]$'.

Notice that we have assumed so far that the left-hand side of an assignment is a simple variable and not a subscripted variable, a procedure identifier or a formal parameter called by name.

LOOPS

We have dealt with the simplest use of assignment, which is to evaluate once and for all an expression which would later occur in several places. Another use of assignment is to update a counter or accumulate a result in a loop. We will translate a loop by a recursive function.

```
      i:=1; x:=0;                 print let i=1; let x=0;
loop: if i>10 then goto out;            let rec loop (i, x)=[i>10→⟨i, x⟩,   |
      x:=x+i×i;                                             |[let x=x+i×i;||
      i:=i+1; goto loop;                                    ||let i=i+1;    ||
out:  outinteger(1, x)                                      ||loop (i, x)   ]]
                                        let ⟨i, x⟩=loop (i, x);
                                        result x
```

The body of the loop has as assignable variables i and x. We need to replace it with a piece of program which produces new values of these assignable variables. The function which produces them will also need to take the old values as parameters. In general

```
loop: if E then goto out;       let rec loop{X}=[E→X,                |
      S{X} goto loop;                                 |[D{X}; loop{X}  ]
out:                            let X=loop{X}
```

We will not consider more elaborate loops as these would need a proper treatment of labels and this is outside the scope of the present paper, which deals with assignment. There seems to be no reason why the present treatment of assignment could not be combined with Landin's treatment of labels (Landin 1966), but this could bear further investigation.

Let us consider **for** statements

for $i:=E1$ **step** $E2$ **until** $E3$ **do** $S\{X\}$

```
let rec loop(⟨i⟩:::X)=
 [let i=E1;                                  |
 |(i=E3×sign(E2)>0→⟨i⟩:::X,                  |
 |[D{x}; loop(⟨i+E2⟩:::X)]                   ]
let ⟨i⟩:::X=loop(⟨E1⟩:::X);
let i=undef
```

Here $D\{x\}$ is the translation of $S\{x\}$ and we have assumed that $E1$, $E2$ and $E3$ have no side effects. If they do have side effects the translation has to be extended to redefine their assignable variables.

There is no difficulty in translating other kinds of **for** statement in a similar manner.

PROCEDURES

Let us see whether we can deal with **procedures** by the methods just elaborated. First we will try a non-type **procedure** with parameters called by value. Suppose that a **procedure** f has as formal parameters a list X and as non-locals a list Y.

Algol	*ISWIM (incorrect translation)*
procedure $f(X)$; **value** X;	**let rec** $f(X)=$
$S\{X:::Y\}$;	$[D\{X:::Y\}; Y]$;
. . . . .	. . . . .
$f(X1)$;	**let** $Y=f(X1)$;
$f(X2)$;	**let** $Y=f(X2)$;

In general this will not work correctly because of the non-locals. This becomes evident if we look at a specific example.

```
procedure f(x); value x;          print let rec f(x)=
  a:=a+x;                               [let a=a+x; result a];
a:=0;                                   let a=0;
f(1);                                   let a=f(1);
f(2);                                   let a=f(2);
outinteger(1, a)                        result a
```

The trouble becomes obvious if we number off the different variables which are all called a. This change in the identifiers used will not affect the meaning of the expression.

```
print let rec f(x)=
        [let a2=a1+2; result a2];
      let a3=0;
      let a4=f(1);
      let a5=f(2);
      result a5
```

The variable $a1$ takes on the value which it has when f is defined and not the value of $a3$ or $a4$ as it would have to in a correct translation of the

Algol text. The point is that the ISWIM function is a quite definite entity which always has the same effect whenever it is used, e.g., if it is the function 'add two' then it always adds two, whereas an Algol **procedure** may add two at one time when it is called and subtract three the next, because there has been an assignment to one of its non-locals meanwhile.

We can deal with the non-locals by letting them appear as extra parameters of the function in the ISWIM translation. Where assignments to some of these non-locals take place we will let the **procedure** produce as a result a list of the values of its assignable variables. This will give us the power to handle non-locals properly without introducing the concept of assignment into the ISWIM translation.

```
procedure f(x); value x;                print let rec f(x, a)=
  a:=a+x;                                     [let a=a+x; result a];
a:=0;                                       let a=0;
f(1);                                       let a=f(1, a);
f(2);                                       let a=f(2, a);
outinteger(1, a)                            result a
```

Algol type **procedures** present no difficulty. If they have side-effects they produce, as well as their obvious result, a list of the new values of their assignable variables.

```
integer procedure f(x); value x;        print let rec f(x, a)=
  begin a:=a+x;                               [let a=a+x;                |
      f:=a×x                                  |result ⟨a×x, a⟩          ];
  end;
a:=0;                                       let a=0;
y:=f(1);                                    let ⟨y, a⟩=f(1, a);
y:=y+f(2);                                  let ⟨e, a⟩=f(2, a);
outinteger(1, a+y)                          let y=y+e;
                                            result a+y
```

Note that where a type **procedure** with side-effects is called as part of an expression, as in '$y+f(2)$' above, it is necessary to decompose the expression and introduce a subsidiary definition. If several such sub-expressions occur the sequence of the subsidiary definitions will correspond to the rule of evaluation chosen, e.g., evaluate from left to right. A rather similar thing occurs when the actual parameters of a **procedure** call are expressions which have side-effects. In this case we must also introduce subsidiary definitions before the call, ensuring that the values of any non-locals being fed to the **procedure** are updated by these definitions.

In this method of dealing with non-local variables we must know when we are translating a **procedure** call which variables are non-locals of the **procedure** and which of these are assignable. There is normally no difficulty about this in Algol, but it is not so easy for a **procedure** f which has another **procedure** g as a formal parameter. In this case the non-local variables of f will differ from call to call and must be taken to include the non-locals of g, similarly

for the assignable variables. This means that we can no longer write the translation for *f* in our previous manner, including the non-locals explicitly as extra parameters. We will return to this difficulty in the section 'Call by reference' (p. 17).

Parameters called by name present another problem. Landin (1965) has shown that they may be treated as functions with no parameters. By this means they can be handled in our scheme by the same method as **procedures** which are parameters of other **procedures**.

DATA STRUCTURES

We now consider the problems presented by assignment in a language which allows processing of data structures. We will take as an example an informal language essentially equivalent to LISP but with an Algol-like syntax. We will use as basic functions HEAD, TAIL and CONS, writing them in capitals to distinguish them from the related ISWIM functions *head*, *tail* and *cons*. The distinction is important, especially in the case of *cons*. The methods we shall use are general enough to extend to other languages which treat data structures.

We will call the language which we are using as an illustration 'List-Algol' and define it by example rather than give an explicit description. No particular virtue is claimed for this language, we are only concerned to find a formalism (other than a pictorial one) which makes the meaning of it clear.

List-Algol	*Explanation*
(i) **begin list** *x*, *y*;	
x:=CONS(1, *nil*);	*x* is a list of one item
y:=CONS(2, *x*);	*y* is a list of two items
HEAD(*y*):=3;	The first item of *y* changes
outinteger(1, HEAD(*y*)+HEAD(*x*))	3+1 is printed
end	
(ii) **begin list** *x*, *y*;	
x:=CONS(1, *nil*);	
y:=CONS(2, *x*);	
HEAD(*x*):=3;	The first item of *x* changes. This also changes the second item of *y*, i.e., HEAD(TAIL(*y*)).
outinteger(1, HEAD(TAIL(*y*)))	3 is printed
end	
(iii) **begin list** *x*, *y*;	
x:=CONS(1, *nil*);	
y:=CONS(*x*, *x*);	*y* is a list with two identical sublists
HEAD(TAIL(*y*)):=2;	This also changes to HEAD(*x*) and HEAD(HEAD(*y*))
outinteger(1, HEAD(HEAD(*y*)))	2 is printed
end	

(iv) **begin list** x;
  x:=CONS(1, *nil*);
  TAIL(x):=x; — x is now a circular list
  outinteger(1, HEAD(TAIL(TAIL(x)))) — 1 is printed
end

Of course it is tempting to simply translate CONS by *cons*, HEAD by *head* and TAIL by *tail*, but this will not do. The following examples show how this naive approach breaks down. To carry out the translations we will need a couple of auxiliary functions defined thus:

let *updatehead*(x, i)=*cons*(i, *tail*(x))
and *updatetail*(x, i)=*cons*(*head*(x), i);

List-Algol	*ISWIM (naive translation)*
(i) **begin list** x, y;	**print let** x=*undef* **and** y=*undef*;
x:=CONS(1, *nil*);	**let** x=*cons*(1, *nil*);
y:=CONS(2, x);	**let** y=*cons*(2, x);
HEAD(y):=3;	**let** y=*updatehead*(y, 3);
outinteger(1, HEAD(y)+HEAD(x))	**result** *head*(y)+*head*(x)
end	

This translation is adequate. Both programs print '2+1'

(ii) **begin list** x, y;	**print let** x=*undef* **and** y=*undef*;
x:=CONS(1, *nil*);	**let** x=*cons*(1, *nil*);
y:=CONS(2, x);	**let** y=*cons*(2, x);
HEAD(x):=3;	**let** x=*updatehead*(x, 3);
outinteger(1, HEAD(TAIL(y)))	**result** *head*(*tail*(y))
end	

This translation fails. The List-Algol prints '3' but the ISWIM prints '1' since it does not cater for the side-effect on y of the assignment to x.

(iii) **begin list** x, y;	**print let** x=*undef* **and** y=*undef*;
x:=CONS(1, *nil*);	**let** x=*cons*(1, *nil*);
y:=CONS(x, x);	**let** y=*cons*(x, x);
HEAD(TAIL(y)):=2;	**let** y=*updatehead*(*tail*(y), 2);
outinteger(1, HEAD(HEAD(y)))	**result** *head*(*head*(y))
end	

This translation fails. The List-Algol prints '2' but the ISWIM prints '1', again because it does not cater for a side-effect.

(iv) **begin list** x;	**print let** x=*undef*;
x:=CONS(1, *nil*);	**let** x=*cons*(1, *nil*);
TAIL(x):=x;	**let** x=*updatetail*(x, x);
outinteger(1, HEAD(TAIL(TAIL(x))))	**result** *head*(*tail*(*tail*(x)))
end	

The List-Algol prints '1', but the ISWIM result is undefined.

REPRESENTING DATA STRUCTURES BY GRAPHS

In our naive translations above we assumed that the correct translations of CONS, HEAD and TAIL were the ISWIM functions *cons*, *head* and *tail*. So long as we are translating a language analogous to pure (functional) LISP this is sufficient, but it is not difficult to see that once we allow assignment to components of data structures, e.g., 'TAIL(x):=y', this translation is no longer adequate. The reason is that *cons*, *head* and *tail* are functions acting on trees, whilst the structures dealt with in a list processing language with assignment to components are much more complex objects than trees. In fact they are a certain kind of directed graph.

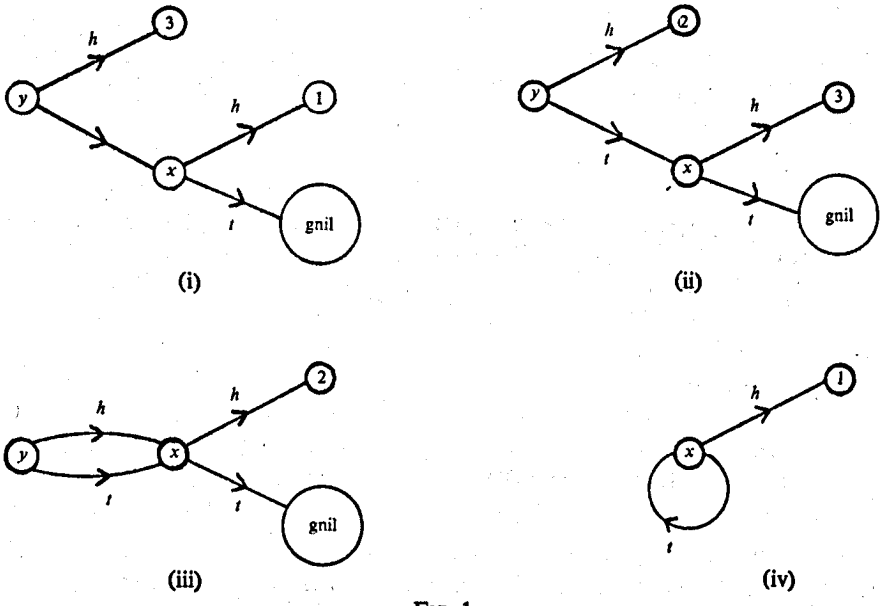


FIG. 1

Such a graph can be represented as a (potentially infinite) set of nodes N, a set of data objects (atoms) A, which might be character strings or integers, and a terminal object which we will call *gnil*, together with two functions h and t from N to $N \cup A \cup \{gnil\}$. We use the word 'item' for elements of $N \cup A \cup \{gnil\}$. The functions h and t are partial and defined only for a subset of the nodes N. Thus an arc in the graph corresponds to applying h or t to a node, and there are two kinds of arc corresponding to h and t respectively. Since N and A are fixed it will be sufficient to represent the graph by the pair $\langle h, t \rangle$ in our ISWIM versions of List-Algol programs. A list variable will have as its value a node in such a graph. The nodes correspond to addresses in a computer and the functions h and t find the contents of the address N.

Fig. 1 shows pictorially the structures produced in the four sample programs of the last section.

We will need some functions to act on these graphs.

The function *gcons* takes two arguments which are any items, i.e., nodes, atoms or *gnil*, and also a graph as argument. It produces a new node and a new graph, i.e., an updated h and t.

```
let gcons(i1, i2, h, t)=[let n3=anyfreenode          |
                        |let h(n)=n=n3→i1, h(n);|
                        |let t(n)=n=n3→i2, t(n); |
                        |result ⟨n3, h, t⟩          ];
```

We write here '**let** $n3=$*anyfreenode*' meaning any node of N for which h and t are not defined. This could be done more correctly by introducing a list of free nodes as an extra parameter of *gcons*. However, this would complicate the expressions without adding anything of real interest. We will simply assume in simplifying the ISWIM expressions that the new nodes introduced are all distinct.

There are two updating functions which produce just a new graph.

```
let gupdatehead(n1, i2, h, t)=[let h(n)=n=n1→i2, h(n); |
                              |result ⟨h, t⟩              ]
and gupdatetail(n1, i2, h, t)=[let t(n)=n=n1→i2, t(n);  |
                              |result ⟨h, t⟩              ];
```

We also need functions for testing for atoms and *gnil*.

```
let gatom(x)=x∈A and gnull(x)= x=gnil;
```

We will use the functions h_0 and t_0 for the null versions of h and t which are undefined for all nodes.

```
let h0(n)=undef and t0(n)=undef;
```

We can now make a more satisfactory translation of our previous examples.

List-Algol	*ISWIM*
(i) **begin list** x, y;	**print let** $x=$*undef* **and** $y=$*undef*
	and $h=h_0$ **and** $t=t_0$;
x:=CONS(1, *nil*);	**let** ⟨x, h, t⟩=*gcons*(1, *gnil*, h, t);
y:=CONS(2, x);	**let** ⟨y, h, t⟩=*gcons*(2, x, h, t);
HEAD(y):=3;	**let** ⟨h, t⟩=*gupdatehead*(y, 3, h, t);
outinteger(1, HEAD(y)+	**result** $h(y)+h(x)$
HEAD(x))	
end	

Substituting the definitions of *gcons* and *gupdatehead* and then making the obvious simplifications we obtain successively the left-hand and right-hand versions below. (We change $n3$ to $n4$ in the second *gcons* to avoid clashes.)

```
let x=undef and y=undef
    and h=h0 and t=t0;
let ⟨x, h, t⟩=
  [let n3=anyfreenode;      |     let n3=anyfreenode;
  |let h(n)=n=n3→1, h(n);  |     let h(n)=n=n3→1, undef;
```

```
  |let t(n)=n=n3→gnil, t(n);|        let t(n)=n=n3→gnil, undef;
  |result ⟨n3, h, t⟩       ];       let x=n3;
let ⟨y, h, t⟩=
  [let n4=anyfreenode;     |        let n4=anyfreenode;
  |let h(n)=n=n4→2, h(n);  |        let h(n)=n=n4→2, h(n);
  |let t(n)=n=n4→x, t(n);  |        let t(n)=n=n4→x, t(n);
  |result ⟨n4, h, t⟩       ];       let y=n4;
let ⟨h, t⟩=
  [let h(n)=n=y→3, h(n);   |        let h(n)=n=y→3, h(n);
  |result ⟨h, t⟩           ];
result h(y)+h(x)                    result h(y)+h(x)
```

Further substitutions give

```
let n3=anyfreenode;
let n4=anyfreenode;
let t(n)=n=n4→n3, n=n3→gnil, undef;
let h(n)=n=n4→3, n=n4→2, n=n3→1, undef;
result h(n4)+h(n3)
```

This has value 3+1 (cf. Fig. 1(i)). Note that the second conditional clause in the definition of h is redundant, i.e., we would simplify to

```
let h(n)=n=n4→3, n=n3→1, undef
```

Note that we could have made the same simplifications if the integers 1, 2 and 3 had been replaced by expressions involving variables or by formal parameters. In other words we have a means of simplifying programs without actually running them to evaluate the result.

We may now translate the second example which involves a side-effect.

List-Algol	*ISWIM*
(ii) **begin list** x, y;	**print let** $x=undef$ **and** $y=undef$ **and** $h=h_0$ **and** $t=t_0$;
x:=CONS(1, *nil*);	**let** $\langle x, h, t\rangle = gcons(1, nil, h, t)$;
y:=CONS(2, x);	**let** $\langle y, h, t\rangle = gcons(2, x, h, t)$;
HEAD(x):=3;	**let** $\langle h, t\rangle = gupdatehead(x, 3, h, t)$;
outinteger(1, HEAD(TAIL(y)))	**result** $h(t(y))$
end	

This reduces eventually to

```
let n1=anyfreenode and n2=anyfreenode;
let h(n)=n=n1→3, n=n2→2, undef
and t(n)=n=n2→n1, n=n1→gnil, undef;
result h(t(n2))
```

This has value 3 as required (cf. Fig. 1(ii)).

(iii) **begin list** x, y;	**print let** $x=undef$ **and** $y=undef$ **and** $h=h_0$ **and** $t=t_0$;
x:=CONS(1, *nil*);	**let** $\langle x, h, t\rangle = gcons(1, gnil, h, t)$;

y:=CONS(x, x);	**let** $\langle y, h, t\rangle = gcons(x, x, h, t)$;
HEAD(TAIL(y)):=2;	**let** $\langle h, t\rangle = gupdatehead(t(y), 2, h, t)$;
outinteger(1, HEAD(HEAD(y)))	**result** $h(h(y))$
end	

This reduces after some substitutions to

let $t(n) = n = n2 \rightarrow n1,\ n = n1 \rightarrow gnil,\ undef$
and $h(n) = n = n1 \rightarrow 2,\ n = n2 \rightarrow n1,\ undef$;
result $h(h(n2))$

This has value 2 as required.

(iv) **begin list** x;	**print let** $x = undef$
	and $h = h_0$ **and** $t = t_0$;
x:=CONS(1, *nil*):	**let** $\langle x, h, t\rangle = gcons(1, gnil, h, t)$;
TAIL(x):=x;	**let** $\langle h, t\rangle = gupdatetail(x, x, h, t)$;
outinteger(1, HEAD(TAIL	**result** $(h(t(t(x))))$
(TAIL(x))))	
end	

On reduction

let $n1 = anyfreenode$;
let $h(n) = n = n1 \rightarrow 1,\ undef$;
let $t(n) = n = n1 \rightarrow n1,\ n = n1 \rightarrow gnil,\ undef$;
result $h(t(t(n1)))$

This has value 1 as required since $t(n1) = n1$.

Before leaving the subject of data structures we will say something about arrays in Algol. These can easily be handled by letting each array correspond to a sequence of functions.

Algol	*ISWIM*
begin integer array a[1:10];	**let** $a(i) = undef$;
a[3]:=1;	**let** $a(i) = i = 3 \rightarrow 1,\ a(i)$;
a[4]:=2+a[3];	**let** $a(i) = i = 4 \rightarrow 2 + a(3),\ a(i)$;

We should really make provision for checking that the subscript is within the bounds. To do this we would need to carry these bounds along with the ISWIM function.

CALL BY REFERENCE

The method of handling data structures described above could be extended to languages in which more elaborate structures can be created. To do this we would introduce not just two functions h and t, but as many as are needed to correspond to the selector operations available in the language in question. What of the case where we introduce only one function, say c for 'contents' instead of h and t? We can use this to explain call by 'reference' or by 'simple name'.

We will modify Algol again to include variables of type **reference**, calling

the resulting language 'Ref-Algol'. We give an example with a naive ISWIM translation, which will be seen to be inadequate.

Ref-Algol	*Naive ISWIM*
begin reference integer u, v;	**print let** $u=$*undef* **and** $v=$*undef*;
procedure $f(x)$;	**let** $f(x, u)=$[**let** $x=x+1$; |
reference integer x;	|**let** $x=x+u$; |
begin $x:=x+1$;	|**result** x];
$x:=x+u$	
end;	
$u:=0$; $v:=0$;	**let** $u=0$; **let** $v=0$;
$f(v)$;	**let** $v=f(v, u)$;
$f(u)$;	**let** $u=f(u, u)$;
outinteger(1, $u+v$)	**result** $u+v$
end	

The Ref-Algol **procedure** increments the variable fed to it as a parameter and then adds u to it. The ISWIM translation fails if this variable is itself u, adding the unincremented value of u instead of the more recent one. We can get a correct translation using our data structure techniques and the function c. We will need functions *gconsref* corresponding to *gcons*, and *gupdatecont* corresponding to *gupdatehead.*

begin reference integer u, v;	**print let** $u=$*anyfreenode*
	and $v=$*anyfreenode* **and** $c=c_0$;
procedure $f(x)$;	**let** $f(x, u, c)=$
reference integer x;	[**let** $c=$*gupdatecont*$(x, c(x)+1, c)$; |
begin $x:=x+1$;	|**let** $c=$*gupdatecont*$(x, c(x)+c(u), c)$; |
$x:=x+u$	|**result** c];
end;	
$u:=0$;	**let** $c=$*gupdatecont*$(u, 0, c)$;
$v:=0$;	**let** $c=$*gupdatecont*$(v, 0, c)$;
$f(v)$;	**let** $c=f(v, u, c)$;
$f(u)$;	**let** $c=f(u, u, c)$;
outinteger(1, $u+v$)	**result** $c(u)+c(v)$
end	

This will reduce to

let $u=$*anyfreenode* **and** $v=$*anyfreenode* **and** $c=c_0$;
let $f(x, c)=$
[**let** $c(n)=n=x\rightarrow c(x)+1, c(n)$; \|
\|**let** $c(n)=n=x\rightarrow c(x)+c(u), c(n)$;\|
\|**result** c];

let $c(n)=n=u\rightarrow 0, c(n)$;
let $c(n)=n=v\rightarrow 0, c(n)$;
let $c=f(v, u, c)$;

let $c=f(u, u, c)$;
result $c(u)+c(v)$

The definition of f reduces further to

let $f(x, c)=[$**let** $c(n)=n=x\rightarrow c(x)+1+[u=x\rightarrow c(x)+1, c(u)], c(n);|$
$\quad|$**result** c $]$

When this is applied to u, '**let** $c=f(u, u, c)$' becomes

let $c(n)=n=u\rightarrow c(u)+1+[u=u\rightarrow c(u)+1, c(u)], c(n)$

i.e.,

let $c(n)=n=u\rightarrow c(u)+1+c(u)+1, c(n)$

which has the desired effect of adding 2 to twice the contents of u.

The application of this method of translation to more elaborate cases should be fairly obvious. We have assumed that variables being used as **reference** variables are declared as such. Even if this were not the case it is only necessary to note which variables are ever used as an actual parameter correspond to a formal parameter called by **reference** and note such variables as **reference** variables.

In our discussion of **procedures** in a previous section we saw that a **procedure** f which has another **procedure** g as a formal parameter presents difficulties if the actual parameters corresponding to g have non-local variables. This is an awkward case where we cannot explicitly provide for side-effects when translating because they vary from call to call of f. We can overcome the difficulty by treating any variables which occur as non-locals of the actual parameter **procedures** as **reference** variables. All we need to do in this case is to feed to f and g the function c which gives information about all such variables.

The function c here plays a similar role to Strachey's store. Strachey (1965) defines an entity σ, the store, and two other kinds of entities, 'left-hand values', α (generalised addresses) and 'right-hand values', β (the contents of the addresses). The updating operator, U, and the contents operator, C, act on these.

Loading from the store is represented by

$\beta=C(\alpha, \sigma)$ (rewriting Strachey's equation on p. 203 in a rather more conventional notation)

and updating the store by

$\sigma'=U(\alpha, \beta', \sigma)$

where σ' is the new store and β' is a bitstring of the same length as σ.

For example $y:=2+x$ transforms σ into σ' where $\sigma'=U(\alpha_y, 2+C(\alpha_x, \sigma), \sigma)$. Now it would appear that the axioms which completely characterise the operators C and U are:

(i) For all α, β, σ

$C(\alpha, U(\alpha, \beta, \sigma))=\beta$

(ii) For all α, α', β, σ, if $\alpha \neq \alpha'$ then
$C(\alpha, U(\alpha', \beta, \sigma)) = C(\alpha, \sigma)$.

These axioms are satisfied by the functions

$C(n, c) = c(n)$
$U(n, i, c) = \lambda n'.\ n' = n \rightarrow i, c(n')$

Where the n is equivalent to Strachey's α and the i to his β. Thus the function c, used as I have used it above, gives a particular model (a very simple one) which satisfies the axioms for Strachey's 'contents' and 'update' functions. The functions h and t used above would be a model for the obvious extension of Strachey's system to lists.

ACKNOWLEDGEMENTS

Much of this work was done while I was on a temporary assignment to the CPL project under the direction of Christopher Strachey. I am very grateful for many instructive and stimulating discussions with him and with Peter Landin. They are not responsible for mistakes or misapprehensions in this paper. My thanks are also due to Donald Michie for never-failing support and encouragement.

REFERENCES

Church, A. (1941). The calculi of lambda-conversion. *Annals of Mathematical Studies*, No. 6. Princeton, N.J.: Princeton University Press.

Curry, H. B., & Feys, R. (1958). *Combinatory Logic*, vol. 1. Amsterdam: North-Holland.

Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer J.*, **6**, 308-320.

Landin, P. J. (1965). A correspondence between ALGOL 60 and Church's Lambda-notation. *Communs Assn Comput. Mach.*, **8**, 89-101, 158-165.

Landin, P. J. (1966a). The next 100 programming languages. *Communs Assn Comput. Mach.*, **9**, 157-166.

Landin, P. J. (1966b). A formal description of ALGOL 60. *Formal Description Languages for Computer Programming*. T. B. Steel, Jr. (ed.). Amsterdam: North-Holland.

McCarthy, J. *et al.* (1962). *LISP 1·5 Programmer's Manual.* Cambridge, Mass.: MIT.

McCarthy, J. (1963). A basis for a mathematical theory of computation. *Computer Programming and Formal Systems.* P. Braffort and D. Hirschberg (eds.), pp. 33-70. Amsterdam: North-Holland.

Rosenbloom, P. (1950). *The Elements of Mathematical Logic.* New York: Dover.

Strachey, C. (1965). Towards a formal semantics. *Formal Description Languages for Computer Programming*. T. B. Steel, Jr. (ed.). Amsterdam: North-Holland.

2

SOME TRANSFORMATIONS AND STANDARD FORMS OF GRAPHS, WITH APPLICATIONS TO COMPUTER PROGRAMS*

D. C. COOPER
CARNEGIE INSTITUTE OF TECHNOLOGY

INTRODUCTION: PROGRAMS AND DIRECTED GRAPHS

In Cooper (1967) I reported on several attempts at proving theorems about computer programs, with the ultimate goal in mind of providing mathematical proofs that programs are correct rather than just testing them on some particular data sets. In the last section I commented that the proofs I obtained in Cooper (1966a) could be viewed as deep results about small programs but that what was needed were techniques for proving shallow results about large programs. By this latter I meant that the proofs did not depend too much on properties of the basic functions or commands used in the program but rather depend on the synthesis of a large number of trivial properties, the particular way this synthesis is performed being guided by the structure of the large program. Some of the other work reported on in Cooper (1967), particularly the work of Evans (1965), has this flavour but are particular proofs about particular programs. Mention should also be made of the long proof of Balzer (1966) which proves that a particular finite automaton correctly performs its given task.

There are many programs whose control structure can be well shown by a directed graph—that is the flow chart of the program. The first goal of this research is to produce some kind of automatic scheme which will prove results about programs which only use conditional statements and assignment statements and we shall not consider those features of a programming

* This work was supported by the Advanced Research Projects Agency of the U.S. Department of Defense under the contract SD-146 to the Carnegie Institute of Technology.

language which imply some kind of control structure not immediately so representable; for example, the passing of parameters to sub-routines, perhaps by the use of procedures and functions previously defined by the programmer. These are very important features of programming languages, but they will not be further considered in this chapter. Although the techniques of this chapter may not be directly applicable in these circumstances, some modification of the ideas used could well be useful; for example, the graphs could represent the relations between a set of mutual recursive function definitions and the transformations to be defined later could well be meaningful.

Having associated a graph with a program, there may well be transformations which can be made on the graph which 'do not affect' the program, later we shall amplify this remark. The question then arises as to whether the graph may be put into some standard form by a series of such transformations, this standard form being one in which it is easier to prove results about the program by taking advantage of special features of the standard form. The purpose of this chapter is to define several such transformations and two such standard forms, and to give results and conjectures about the possibility of transforming programs to the standard forms.

The particular way in which a directed graph is associated with a given program is not important. All that matters is that the graph transformations defined in the next section should represent meaningful transformations if performed on the programs with which the graphs are associated. However, in order to better illustrate the intention behind the transformations, we give a particular definition of a program and of the graph associated with a program. This definition will use a state vector approach (see McCarthy 1960). In this approach the current state of a computation is represented by the value of a vector, each component of which corresponds to one of the variables, machine locations, etc. (dependent on the particular programming language used). The effect of a basic command can then be described by a function whose argument is a state vector (the state of the machine before the command is obeyed) and whose value is also a state vector (the state of the machine after the command is obeyed). A sequence of commands corresponds to composition of the corresponding functions. A two-way test is represented by a predicate whose argument is a state vector.

Assume then that we have a domain D, a set of functions f_i which map D into D and a set of predicates p_i whose domains are D. A *program* is defined to be a directed graph with every arc of which is associated one of the f functions and one of the p predicates. It is a *legal program* if the graph has just one node (A, say) with no arcs leading to it, if it has just one node (Z, say) with no arcs leading from it, if every node is on a path which starts at A and ends at Z, and if for all nodes (except Z) the set of predicates on all arcs leading from the node is complete and mutually exclusive.

The intended interpretation is that if f_i and p_j are the function and predicate associated with some arc then f_i represents the effect of all the statements obeyed along that arc and p_j represents the condition that the arc is entered.

A program then defines a partial function from D into D in an obvious manner (partial because the program may loop indefinitely).

It should again be emphasised that we are not concerned with this particular definition of a program, only with the ability to associate a graph with a program. As alternatives we could have associated the information in the program with the nodes rather than the arcs, or we could have taken some simple programming language and explicitly stated how to obtain the graph from the program.

The results to be obtained later could well have applications in other areas than computer programs. For example, Seshu and Reed (1961) describe the application of graphs to sequential machines and to systems of linear algebraic equations. All our transformations can be given interpretations in these areas, in fact the delete node transformation (T 5 of the next section) is already well known in these applications. The particular field of interest determines the transformations of interest; for example, later we shall reject a certain transformation (DT 1) as not being a useful one for computer programs. However, this transformation is of use in the areas of sequential machines and of linear equations.

EXAMPLES OF TRANSFORMATIONS

In this section we define several directed graph transformations of interest. In each case we give a description of the general case in English and illustrate the transformation with a particular case. The illustration of course only shows how the affected part of the graph is altered, the graph may have other nodes and arcs except that in all cases we assume that there are no further arcs which lead into or out of the nodes labelled N.

T 1. Stretch a node N by outputs.

Let O be some subset of the output nodes of N, this subset may be all the output nodes but must not be empty. Delete all arcs from N to nodes in O, add a new node N_1 and add arcs from N to N_1 and from N_1 to all nodes in O. (The output nodes of a node N are all those nodes P such that there is an arc NP, including N itself if there is an arc NN.)

In the example O is the set {N, D, E}.

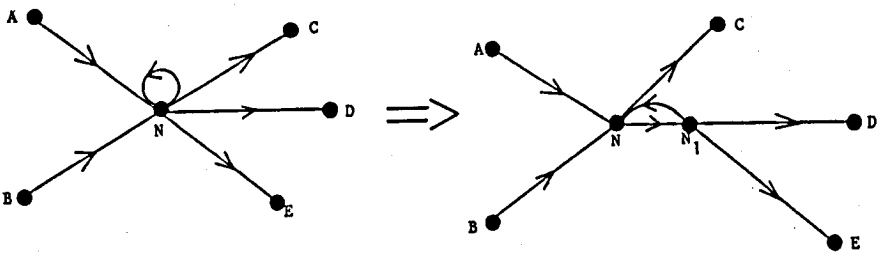
T 2. Stretch a node N by inputs.

Let I be some subset of the input nodes of N; this subset may be all the input nodes but must not be empty. Delete all arcs from nodes in I to N, add a new node N_1 and add arcs from N_1 to N and from all nodes in I to N_1. (The input nodes of a node are defined in an analogous way to the output nodes.)

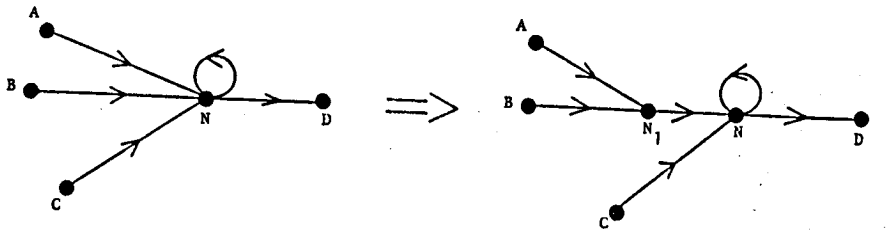
In the example I is the set {A, B}.

T 3. Duplicate a node N.

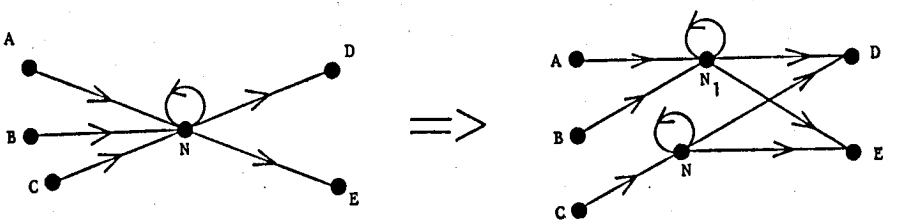
Let I be some non-empty, proper subset of the input nodes of N, not including N itself if NN is an arc. Delete all arcs from nodes in I to N, add a new node N_1, add arcs from N_1 to all the output nodes of N (except N itself if NN is an arc) and add arcs from every node in I to N_1. If NN is an arc also add an arc N_1 N_1.



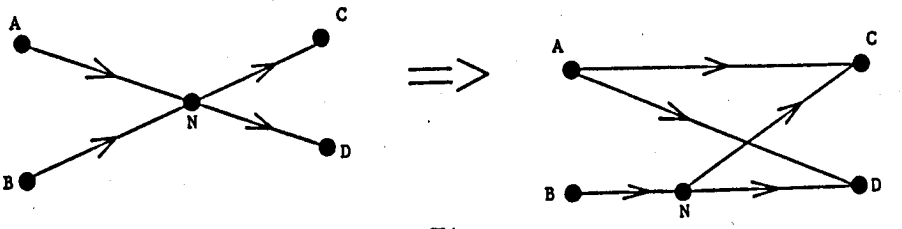
T1



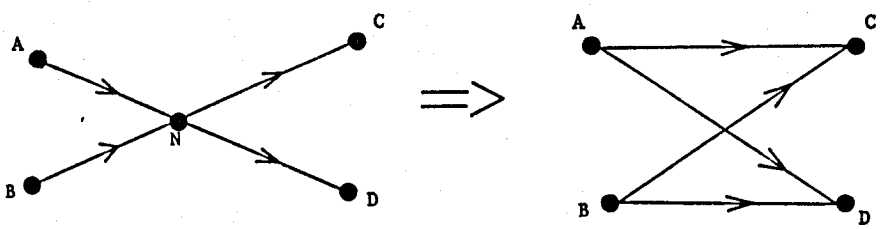
T2



T3



T4



T5

Transformations

In the example I is the set {A, B}.

T 4. Delete an arc AN.

This transformation may only be made if there is no arc NN. Delete the arc AN and add arcs from A to all the output nodes of N. If A was the only input node of N then also delete node N and all arcs leading from N.

T 5. Delete a node N.

This transformation may only be made if there is no arc NN and is equivalent to applying a sequence of T 4 transformations in any order to all the arcs leading to N. Add all possible arcs from an input node of N to an output node of N and delete node N and all arcs leading to or from N.

In all these cases if the untransformed graph is the graph of some program P it is easy to find an equivalent program with the transformed graph as its associated graph. For example, in the diagram illustrating transformation T 1 if $p_{AN}(s)$ and $f_{AN}(s)$ are the predicate and function associated with arc AN on the untransformed graph (and similarly for the other arcs) then in the transformed graph on arc NN_1 we have the predicate $p_{ND}(s) \vee p_{NE}(s) \vee p_{NN}(s)$ and the identity function, N_1N, N_1D and N_1E have the same predicates and functions as NN, ND and NE respectively, and all other arcs have the same predicates and functions as before the transformation. Transformation T 4 represents the effect of anticipating the test at N at the previous node A so that on the transformed graph, for example, with the arc AC we associate the predicate $p_{AN}(s) \wedge p_{NC}(f_{AN}(s))$ and the function $f_{NC}(f_{AN}(s))$.

As two examples of transformation which we do not wish to allow consider the following:

DT 1. Delete a node N with a loop.

This is defined exactly as T 5 but we remove the restriction that there must be no arc NN.

DT 2. Join two arcs AB and CD.

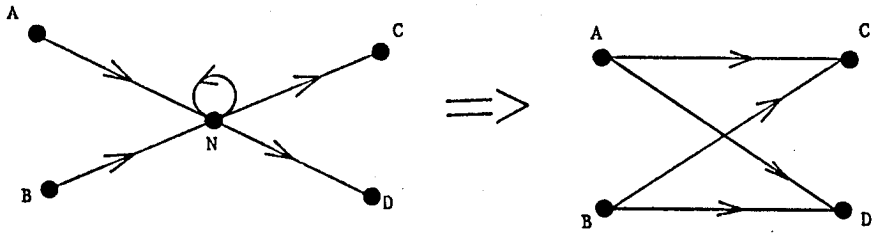
Delete the arcs AB and CD, add a new node N and add arcs AN, NB, CN and ND.

In DT 1 with arc AC we would have to associate a predicate expressing the condition that when control finally comes out of the loop at N control passes to node C. This can be done, but in general requires an infinite disjunction, we assume this is not a useful concept to introduce. The aim in making the transformation is to better reveal the program's structure, we do not wish to produce a transformed program which introduces complex new concepts and makes our task of proving the desired theorem about the program much more difficult. Admittedly we are being vague here, the transformations T 1 to T 5 imply 'we do not mind complicating our programs by forming conjunctions and disjunctions of predicates and compositions of functions'. Note that the situation arising in DT 1 would not be improved by adding loops on the transformed graph at A and B, or at new nodes internal to the arcs AC, AD, BC and BD.

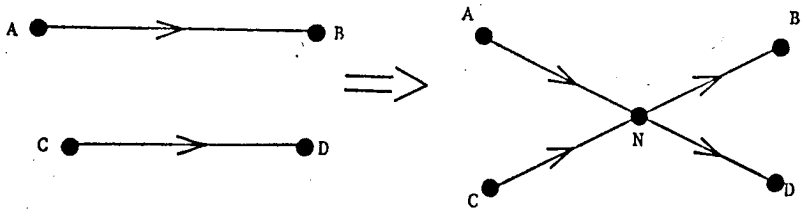
We could allow the transformation DT 2 if an extra Boolean variable is

defined and semantics added to the program to set this variable true along arc AN, false along arc CN. The predicate on arc NB would then be a test whether this variable is true, on arc ND whether it is false. In effect we are coding the path already taken on the graph into a variable. Again we assume that this is not a good thing to do.

In this chapter we define T 1 to T 5 as the only allowable transformations, and then prove or conjecture theorems concerning them. We would like to somehow characterise the allowable transformations and perhaps produce



DT1



DT2

some more. A possible approach is to define a transformation of one graph into another by a definition of the following form:

$G \rightarrow G'$ if and only if $(p)(\exists p')[G = \text{graph}(p) \supset \{G' = \text{graph}(p') \wedge f(p, p')\}]$.

This definition presupposes that we already have a definition of a program, of a function 'graph' which produces the graph of a program and of a predicate $f(p, p')$. This predicate must express the condition that program p' is equivalent to program p, and moreover can be produced from it with some given rules which allow such things as function composition but does not allow anything corresponding to the infinite disjunction of DT 1 or the new Boolean variable of DT 2.

PROPER CYCLE FREE FORM

Having defined the allowable graph transformations, are there some particularly desirable standard forms into which we can try to transform a given graph? Two desirable forms will be defined, one in this section and one in the following section.

Most, if not all, of the questions we wish to answer about a computer program are easy to answer if there are no cycles in the program. The simpler the interconnections between the cycles the easier it should be to

answer questions about whether a program performs correctly; it should therefore be profitable to make transformations aimed at simplifying the cycle structure of a graph.

The simplest (and also rarest) program is one which has no cycles. Next one might consider programs in which all the cycles are independent, i.e., no section of the program is in more than one cycle. This class of programs is more interesting but still very restrictive. However, experience indicates that a large number of interesting programs may be transformed by a series of transformations into a program with this independent loop structure.

In order to make these vague remarks more exact we define a *proper cycle* of a graph to be a cycle through two or more nodes and a *proper cycle free* (PCF) graph to be a graph with no proper cycles. A graph which is PCF clearly has the independent looping property, and it is also trivial to transform any graph with independent loops into PCF form. We therefore take PCF form as being a standard form into which we try to transform a given graph, and obtain necessary and sufficient conditions for a graph to be transformable into PCF form together with an algorithm for making the transformation if it is possible. For this purpose only transformation T 5, delete a node, turns out to be relevant. If a graph can be transformed to PCF form by a sequence of transformations, each of which is one of T 1 to T 5, then this may be done by a sequence of T 5 transformations alone.

Consider graph 1. By deleting node A we obtain graph 2 which is not PCF (and which in fact cannot be transformed to PCF form at all; see the theorem below). But if instead of deleting A we delete B and then C we obtain graph 3 which is PCF and 'obviously much simpler' than graph 2. (The 2's on graph 3 indicate multiple arcs.) Graph 3 reveals the 'basic structure' of graph 1 in a way which graph 2 does not. These graphs show that the order in which nodes are deleted is critical.

Define a graph to be *loop connected* if it contains two nodes, A and B say, and three cycles α, β and γ such that: (i) there is no one node which is on α, β and γ, (ii) A is on α but not on β, (iii) B is on β but not on α, and (iv) both A and B are on γ. This is a generalisation of part of graph 2 in which arcs have been replaced by paths. Then we have:

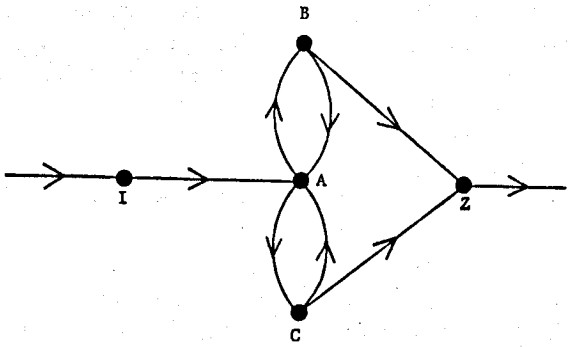
Theorem 1. A graph G can be transformed by a series of transformations, each of which is one of T 1 to T 5, to a graph which is PCF if and only if it is not loop connected.

In Cooper (1966b) only transformation T 5 is considered, but it is easy to extend the results of that paper in such a way that Theorem 1 is an immediate deduction from the theorems proved there.

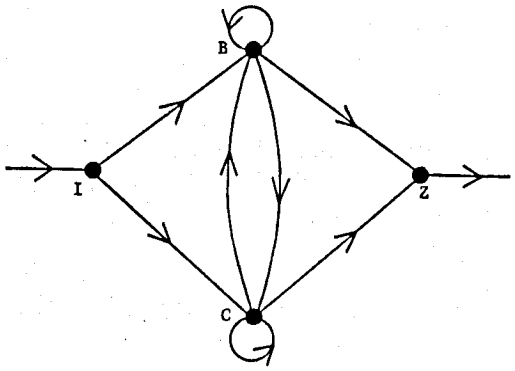
Theorem 1 does not give an algorithm for performing the sequence of transformations. In order to do this define a delete node N transformation (T 5) to be a *safe deletion* if there is some node A (not N) of the untransformed graph with the property that all cycles of the graph which pass

through node N also pass through node A. For example, in graph 1 B or C may be safely deleted but not A. Then the following theorem gives the required algorithm; it is an easy extension of one of the theorems in Cooper (1966b):

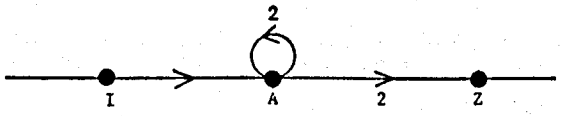
Theorem 2. Start with any graph G and perform safe deletions in any order until a graph G' is obtained on which no safe deletions may be



GRAPH 1



GRAPH 2



GRAPH 3

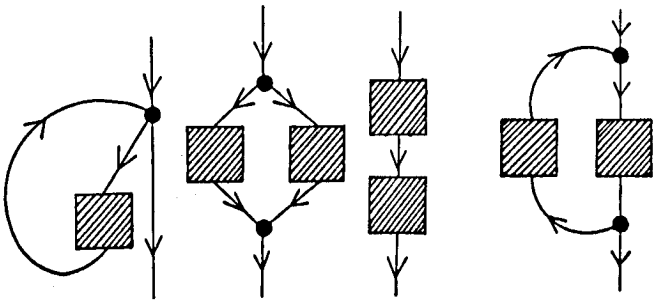
perfórmed. This process must terminate and either G' is PCF or there is no sequence of transformations, each of which is one of T 1 to T 5, which will transform G to a graph in PCF form.

Theorems 1 and 2 completely solve the relevant questions about transformations to PCF form. In the same paper an equivalent condition to loop connectedness is defined; this condition appears more complicated but

simplifies the proofs. In addition a theorem is proved which shows that safe deletions is the widest possible subclass of deletions, i.e., if we make a deletion which is not a safe deletion we obtain a graph which is loop connected and hence not transformable to PCF form.

BLOCK FORM

The second standard form corresponds in a more natural manner than PCF form to the way a large number of programs are written; it roughly corresponds to programs in which all loops are properly nested. A recursive definition of block form is simply given: a graph is defined to be in *block form* if it is of one of the forms B1, B2 or B3 (see the diagrams) where the shaded boxes are either arcs or sub-graphs in block form.



Block form definitions — Alternative to B1

It is not necessary to take exactly these forms; for example, instead of B2 we may allow any number of blocks in parallel and instead of B3 any number in series, obviously a graph of such form can easily be transformed into the standard block form. A more useful equivalent standard form is where instead of B2 and B3 we allow any graph made up of sub-graphs, each in block form, and these sub-graphs joined in any manner that does not create a cycle. Such a graph can be further transformed into the standard block form, but it may well be better not to carry out these further transformations as this usually involves a lot of duplicating of nodes. A third alternative is to replace B1 by B1a.

Very little progress has been made so far with proving results about transformation of graphs to block form, although a large number of small graphs have been investigated. A conjecture has been made as to the necessary and sufficient conditions for a graph to be transformable to block form and we hope soon to have an algorithm to effect the transformations. Any graph in PCF form can be transformed to block form; essentially it already is in block form according to the second alternative definition. However, consider graph 4, which is clearly loop connected and hence cannot be transformed into PCF form. First use T 1 (with N being A and O the set $\{B, Z\}$) and then use T 3 (N being A again and I the set $\{B\}$), this produces graph 5

and it is obvious how to put this into block form. Thus graph 4 is an example of a graph which can be transformed to block form but not to PCF form. As a more complex example, graph 6 may be transformed to block form; this is left as an exercise for the reader; a good start is to delete node A.

What graphs cannot be transformed to block form? Here we have no proven results but it seems almost certain that graph 7 cannot be so transformed (or any graph containing this as a sub-graph). The situation is much more complex than for the PCF case; there a knowledge of just the cycles of a graph was sufficient to determine whether the graph could be transformed to PCF form, but graphs 4 and 7 have the same cycles and one can be transformed to block form but not the other. It seems that a generalisation of graph 7 in which its arcs are replaced by paths (possibly intersecting each other) is the key situation which, if it occurs in a graph, makes the graph untransformable to block form. But examples show that if enough of the paths intersect then we are back to the position of being able to transform to block form.

By induction from examples tried we make the following definition:

A graph is *loop connected with two exits* if it contains three nodes, A, B and Z say, a cycle α through A, a cycle β through B, a path γ connecting A to B, a path δ connecting B to A, a path ϕ connecting A to Z and a path ψ connecting B to Z (see graph 8). Furthermore, there must be no node which is on all three members of any of the following triples:

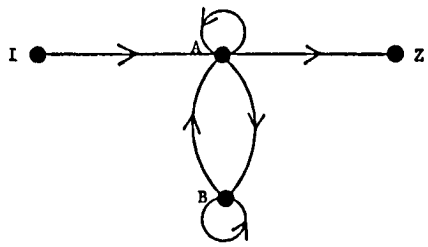
$$(\alpha, \beta, \gamma), (\alpha, \beta, \delta), (\delta, \alpha, \psi), (\gamma, \beta, \phi), (\phi, \psi, \alpha) \text{ and } (\phi, \psi, \beta).$$

We conjecture the following:

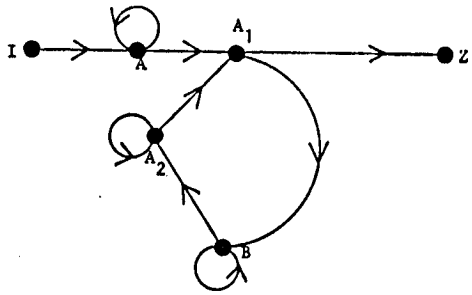
Theorem 3. A graph can be transformed by a series of transformations T 1 to T 5 to a graph which is in block form if and only if it is not loop connected with two exits.*

Böhm and Jacopini (1966) investigate the reduction of flow diagrams to standard forms, they show that corresponding to any flow diagram there is an equivalent one which is 'decomposable into Π, Φ and Δ'. This is precisely block form. Their algorithm for producing this equivalent flow diagram may be looked on as a sequence of graph transformations which, however, allow the transformation DT 2, which we have not allowed. Transformation DT 2 on programs was justified by the introduction of a new variable; rather than introducing a separate new variable every time DT 2 is used, Böhm and Jacopini introduce a single Boolean stack and add semantics to test the top of the stack, to delete the top of the stack and to add a new truth value to the top of the stack. This stack may be thought of as a code for the path through the flow diagram traversed so far. For our purposes we do not wish to allow this transformation, but it is an interesting result that if DT 2 is allowed then any graph may be transformed to block form.

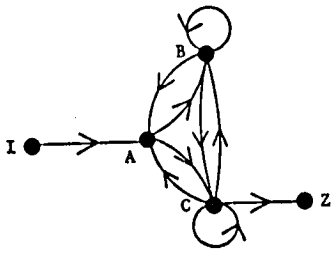
* The theorem is false as stated. It can be corrected by restricting Z to be a node which is not on any cycle. (Added in proof.)



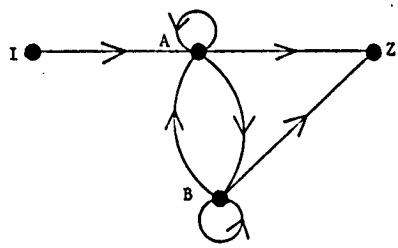
GRAPH 4



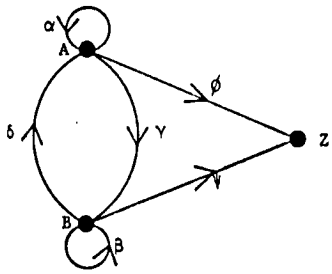
GRAPH 5



GRAPH 6



GRAPH 7



GRAPH 8

REFERENCES

Balzer, R. M. (1966). *Studies concerning minimal time solutions to the firing squad synchronization problem,* Ph.D. thesis. Pittsburgh: Carnegie Institute of Technology.

Böhm, C., & Jacopini, G. (1966). Flow diagrams, Turing machines and languages with only two formation rules. *Communs Assn Comput. Mach.*, **9**, 366-371.

Cooper, D. C. (1966a). The equivalence of certain computations. *Computer J.*, **9**, 45-52.

Cooper, D. C. (1966b). Reduction of programs to a standard form by graph transformation. Presented at 'International Seminar on Graph Theory and its Applications', Rome, Italy, July 5-9, 1966.

Cooper, D. C. (1967). Mathematical proofs about computer programs. *Machine Intelligence* 1, N. L. Collins and D. Michie (eds.), pp. 17-28. Edinburgh: Oliver and Boyd.

Evans, A. (1965). *Syntax analysis by a production language,* Ph.D. thesis. Pittsburgh: Carnegie Institute of Technology.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communs Assn Comput. Mach.*, **3**, 184-195.

Seshu, S., & Reed, M. B. (1961). *Linear Graphs and Electrical Networks.* Massachusetts and London: Addison-Wesley Publishing Co., Inc.

3

DATA REPRESENTATION—THE KEY TO CONCEPTUALISATION

D. B. VIGOR
COMPUTING DEPARTMENT
UNIVERSITY OF GLASGOW

Conceptualisation is the formation and storage of concepts (Hunt 1962). A concept is a general rule for action or a general description of a set of attributes of objects. It can be looked at as the representation of a criterion for set membership whether the set be a set of descriptions or a set of actions. The formation of such descriptions or rules requires a notation in which the relations and modes of action can be represented. This notation should be general enough not to limit too strictly the nature of the concepts that can be described using it.

In this paper I would like to propose a model on which to base such a notation. I hope to indicate how this model can be used to set up a method of programming in which the programmer does not have to know a large amount about the system he is using, and can use primitive concepts much more related to conventional mathematics and to his problem, yet still generate 'efficient' programs.

ALGOL and FORTRAN have made large steps in giving a formal notation for numerical programming, but the primitives used are blunt and clumsy in describing actual computations on natural numbers. The programmer has to know concepts like '*real*', '*integer*', and to redundantly declare the nature of his results although these are 'obvious' from context. All these are irrelevant to the nature of the problem he is solving and are consequences of the fact that we use 'real machines' to carry out our computation.

In 'non-numerical' computing with our present notations we are even more bound by the nature of the machines and systems we use, and a large amount of ingenuity and skill is wasted on fitting problems into moulds which do not

suit them. This leads to frustration on the part of the problem solver, and inefficiency inside the machine.

The main items we wish to handle in computing processes are 'representation of objects' to which we can assign various properties or 'attributes' such as value, structure, permanence, and relationship with other 'objects'. Before any object, defined or undefined, can be considered to exist in a computer we must assign to it a 'name'. The 'name' is the primary attribute of 'existence'. This 'name' need not be specifically stated and can be implied by the method of formation of the object's representation. As 'nameless' objects are not members of our universe of discourse we can use the concept of 'name' to represent the 'object'. We will thus use the two words 'name' and 'object' synonymously because there is an isomorphism between them which is unresolvable.

One or more of the properties or attributes of a name inside a real or virtual computer can be looked at as a 'value' of the name. The 'value' of a name will depend upon the nature of the operation being carried out on that name. The method of representing in the computer a 'value' may or may not be the same as that used for representing 'names'.

A mathematical model of the above situation can be found in the theory of coloured graphs (Scoins 1967). A coloured graph is a set of 'named' points (modelling objects) which are connected by links (modelling relations) which are also 'named'. If we extend the definition of relation to allow for more than 2-argument relations, there is a duality between points and relations which allows us to look at the 'graph' as a set of relations which are linked by the entities over which they hold. A monochromatic sub-graph is a set of points having the same relationship. It is possible conceptually and practically to separate out from a coloured graph a monochromatic sub-graph.

There is one completely general way of representing such 'entities' as coloured graphs inside computers which has been exploited in the programming language LISP. There is a theorem of graph theory (Scoins 1967) which states that any general graph can be represented as a binary ordered directed graph. Directedness can be overcome by introducing dummy 'back' pointers; orderedness is an extra property which need not be considered; and any branching can be represented by two-way branching by introducing 'dummy points'. There are however other representations which are more 'efficient' in particular problem situations.

We can represent the operation of a machine in a static form. This can be done in a number of different ways; (*a*) sequential forms such as programs in ALGOL, FORTRAN, machine code and Ianov schemata (Rutledge 1964); and (*b*) nonsequential forms such as functional expressions, λ expressions (Landin 1964, 1965, 1966) and conditional forms (McCarthy *et al.* 1962). If one accepts a notation for program texts (R. Burstall, pp. 3-20 of this volume) based on Church's lambda-calculus (Church 1941), then one notes that there are basically three types of functions required in static program specification. These are Constructors, Selectors and Predicates.

1. *Constructors* are operations or functions which allow for the construction of compound or complex objects out of simpler ones, e.g., *cons* in LISP, *concatenate* in string processing languages and in Backus Normal Form, *merging* in commercial languages, + and × in arithmetic languages.
2. *Selectors* are functions which find sub-objects from more complex ones, e.g., *car* and *cdr* in LISP, any parsing algorithm, any name in a programming language, $A(i, j)$ in ALGOL arithmetic expressions, 'the verb of this sentence' in English, '**if** *p* **then** . . . **else** . . .' in ALGOL, and 'λ . . . . . .;' in LISP.
3. *Predicates* are functions which map classes of objects on to the truth values **true** or **false.** They can in certain systems, discussed later, be completely handled by special selectors, e.g., **eq**(x, y), **null**(x) in LISP, 'A is a bird' in English (assuming we have a test for birdness).

If we want to talk about evaluation or change resulting from use of functions with specific arguments, it is convenient to distinguish three other types of functions.

4. *Designators* are functions which map 'statements' in one representation onto corresponding 'statements' in another representation. These statements can be either
   (i) descriptions of the allowable states of an object, or
   (ii) descriptions of allowable transformations on objects, i.e. functions.

   Designators may have 'side effects':
   (i) when mapping functions from one representation to another we may restrict, in some way, the class of arguments which can be used with the function;
   (ii) when mapping an object, we may restrict the class of functions which can accept this object as an argument. In these cases we say that the designators changes the domain of validity of the function or object which it is mapping. Some near approaches to designators in existing languages are:

   integer x, **real array** $A[1:n]$ in ALGOL, NIL in LISP, **true** and **false**, **atom.**
5. *Applicators* are functions which specify that a described action, which is the argument of the applicator, has to be carried out within a given designational system, e.g., EVAL . . . in LISP. In most languages the applicator is implied by the program form or is specified externally by pressing a button on a hardware computer. In ALGOL an applicator is **comment** . . .; or in the general purpose macro-generator (Strachey 1965) the macro protection brackets are a type of applicator. In PL1 (I.B.M. Corp. 1965) the operation *compile* is an applicator.
6. *Metafunctions.* These are specific operations which must be loaded on to a computer in order for a 'system' to operate. In LISP these are the constructor *cons*, the selectors *car*, *cdr*, and *cond*, the predicate

eq(p, q), and the designators *true*, *false*, *nil*, *atom* (and their implied applicator *eval*).

I would like to look at the types of designators which we may require in the most general programming situations. In order to do this I need to define a few terms more accurately:

1. *A data structure* is a partially ordered set of items of information, e.g., a character string, a matrix, a tree, an integer, a function in a given representation.
2. *A function* is a mapping of one class of objects onto or into another.
3. *The domain of validity* of a function is the set of ordered pairs of objects for which the function is valid.

$$\text{domain } Z=\{z=(x, y) \mid x\in X \text{ and } y\in Y \text{ and } y=f(x)\}$$

where X is called the input domain, Y is called the range.

4. *A machine feature* is a particular set of hardware or software characteristics of a real or virtual computer, e.g., an adder, a register, a bank of core store, a magnetic tape, a nesting (FILO) store, a buffer (FIFO store).

A machine feature representing a function could be a sequence of changes of state in a particular piece of hardware. The concept of a machine feature involves the idea of predetermined behaviour when given a specific cue or nudge, and corresponds to initiating the evaluation of a function using a particular piece of hardware. The evaluation corresponds to the 'action' of the machine feature.

The orders representing a subroutine when evaluated by a machine initiate changes in state of particular components in this machine. Each one of these components has fixed rules for changing its state, as well as a fixed set of states. The mechanism of initiating a particular sequence of changes, and the components in which these changes occur, are the machine feature corresponding to this particular mechanism.

If we wish to represent an object which is a state or set of states, then the machine feature corresponding to this object is the assemblage of the states, to be set up in specific components of the machine in order to represent this state or set of states. An account of a language for describing objects resembling machine features and their actions has been published recently (Smith 1966).

A designator is a function from data structures on to machine features. I would like to consider the nature of both the function and the domain of validity in certain commonly used designational systems; a designational system is the designator plus a specification of its domain of validity. In non-ambiguous, time-independent computing processes, designators are one-one and onto functions and hence the input domain and the range are of the same cardinality. Thus we can conceive of inverse designators. The idea of non-one-one designators is interesting in dealing with real time

processes, non-deterministic automata and self-adaptive systems, but this is beyond the scope of this paper.

An applicator is a function from the designator of a functional representation onto the 'action' of a machine feature. The 'action' of a machine feature can be formally considered to be the mapping described by applying the 'inverse' of this designator function to the machine feature. It is a selector which selects a data structure corresponding to the particular machine feature. The apparent circularity of this definition is broken by having certain meta-functions which can be looked at either as metadesignators or meta-applicators which carry out the 'actions' required.

In other words, the action of a machine feature is determined by the structure of the machine feature, which can be represented as a formal data structure. We can distinguish some of the following machine features in most machines:

bit, byte, word, vector of words (segments), . . .

and the following data structures are representative of the objects we may wish to describe 'outside' the machine:

booleans, characters, arithmetic quantities (e.g., integer and real), vectors, arrays and trees, general linear graphs, thesauri.

We can relate machine features to one another by describing the constructors which assemble complex ones from simpler ones, and the selectors which allow their subdivision into the basic modules. We can also represent a formally corresponding set of constructors and selectors for the items outside the machine. Now we can see that a designator system can be represented in two formally equivalent ways. One is as a mapping of a set of functions onto another set of functions (this is the compiler approach to the problem), and the other is as a set of data items onto a set of actions of machine features (this is the interpretive approach to the problem).

Let us now look at some of the systems of mapping which have been developed and used till now. In order to simplify this survey, I shall presuppose a result I would really like to demonstrate: that one can satisfactorily classify these systems according to a set of criteria and that my particular choice of hierarchy between these criteria leads to a practically useful model of a system. I distinguish basically three main classes of representation of data which I call the *extended form*, the *mapped form*, and the *coded form* of information. I make the bold statement that the classification of any item into a given representation system is fairly unequivocal.

This classification depends on the assumption that at any level we are considering, data has a structural part (a set of interrelations between items) and a contents part (a group of properties that are purely local to any item). The contents part of a bit is its value 1 or 0, the structure is 'unity'. At this level the contents of an 'integer' could be a 'vector' of bit values in a binary machine, and the structure defined by three parameters: the length of the

integer in bits, the order in which the bits are to be taken, and the class of operators allowed to operate on this 'item'.

In the *extended form*, both the 'structure' and 'contents' are stored within one machine feature. Examples of this are:

1. LISP (McCarthy *et al.* 1962), where the unit is a pure 2-catena or a node in a binary oriented (ordered) graph; the single machine feature is a storage unit (considered to be the top of the stack) which is a function with two arguments, the 'symbol' and the 'link'.
2. IPL5 (Newell *et al.* 1961), which has two 2-argument machine features, one of which accepts a function and a symbol, and the other a link. This can be looked at as a 2-catena with internal structuring.
3. CORAL (Roberts 1965), which is a ring structure language which has a 3-argument machine feature, and a 3-catena as its basic item—one 'symbol' and two 'links' (a forward and a backward link).

In the *mapped form* the 'structure' and the 'contents' are mapped on to separate 'machine features' and could be independently manipulated. Either or both of the structure and the contents can be functional. Examples are:

1. The storage of arrays in 'Kidsgrove ALGOL' (English Electric Leo Marconi 1965), where one has a 'dope vector' which specifies the structure of the array in a form acceptable to one or more selector or 'indexing' functions, and the array contents are stored as a linear, sequential vector in core store (Fig. 1).
2. The 'Multics' (Dennis 1965) file system in which files are stored as segments with heading information on the structure of the file, and segments containing the actual information being stored.
3. A bit map of a 'sparse matrix' in which, associated with every item in the matrix, is a bit which is zero if the element is zero, unity if the element is not, and all the non-zero elements are stored elsewhere as a linear vector of words.

In the *coded form* the structure and the contents are again separated, but either one of these may be rearranged in some way by using some 'artifact' (in the biological sense, i.e., a side-effect) of the representation to handle the information more effectively in a given situation. Examples are:

1. Word condensation in a 'sparse matrix': The matrix contains a large number of sequential zeros when represented in a mapped form like (1) above. We can replace each string of zeros by a single zero followed by the number of such occurrences. The zero can be looked at as a function call using the parameter following it to generate the data concerned.
2. Any syntax-directed programming language has added 'redundancy' which checks the validity of messages transmitted in the language. The constructors for the language must be able to put in this redundancy, a selector or parsing algorithm removes it and checks acceptability of strings.

In any sequence of operations to be carried out using specific machine features there are certain requirements on invariance of domain (i.e., overflow, type clash) which can be built into a system. It would be, however, more natural to allow feasible calculations to be carried out within the specifications given externally without making any assumption about internal representation.

The most common feature of this nature is automatic multilength arithmetic processes where the mathematician *may* not want to be bothered with

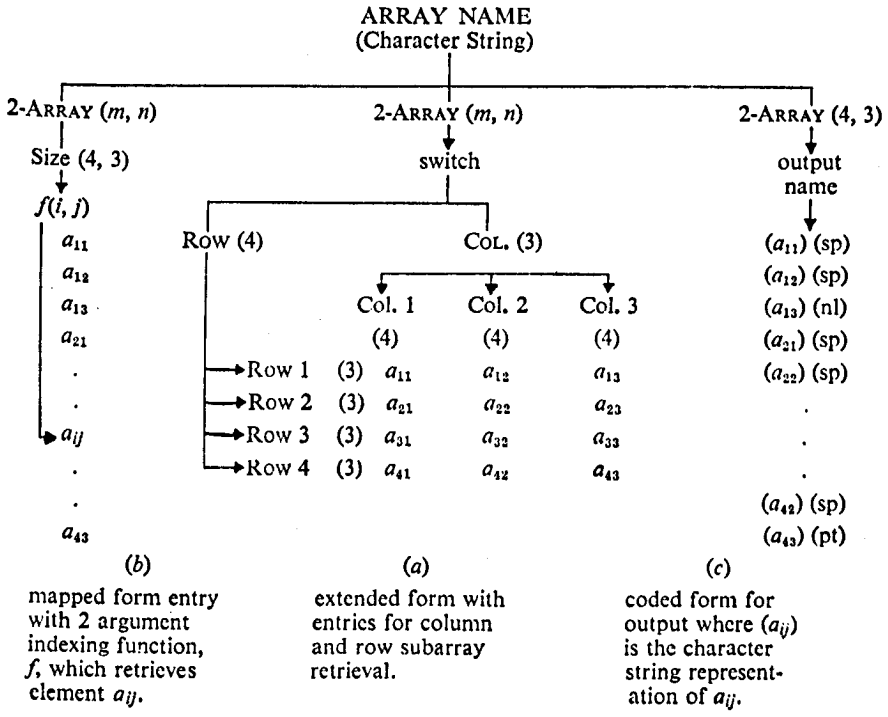


Fig. 1. Example of Array Storage in DDPS (m=4, n=3)

Note: (sp)=space, (nl)=new line, (pt)=page throw.
In all cases the 2-array is treated as a function.

absolute size considerations. Operations on mixed items which are externally compatible, e.g., **integer** and **complex double length integer** gives **complex double length integer** and to do this should require no specific declaration.

It is also possible for machine features to have invariant properties, e.g., an indexing register has certain strict requirements on its domain of 'arguments'. The aim of the systems programmer should be to map the invariants of objects outside machines into invariants of the machine features used in implementing them.

The representational system can be changed during the course of operation of the program—that is, we may choose different 'invariant' properties and

relations in order to apply a new class of functions to the same data. The 'decision' to do this can be left to a program which I shall call a 'system preprocessor', which, during the course of preprocessing a program, 'decides' on the most efficient representation of the problem in this particular machine (where a machine is a set of machine features with specific properties). The decision of efficiency can be made by using the following criteria:

1. 'Best fit' between 'machine feature' invariants and 'data structure' invariants.
2. The 'domain' of the 'data structures' maps onto the 'domain' of 'machine features' with the least 'domain overspill', i.e., with the maximum use of machine feature properties.
3. The 'information loss ratio' (I_l) in the system should be maximal.

$$I_l = t_R / t_M$$

where t_R = time taken on 'information reduction' or processing operations;

t_M = time taken on manipulative orders.

Thus the system should be particularised as early as possible in its 'life' inside the computer.

4. Minimal requirement of extra special purpose machine features in implementation, e.g., one can save time by 'syntactically inverting' transfer functions from one data structure representation to another, thus generating the 'inverse transfer function'. Some work on this is at present being carried out at Glasgow University (Patterson & Vigor 1967).

In the *mapped form* one can carry out subclassification of the properties of the constructor and selector functions, and hence of the structural part of data structures, according to the following criteria:

1. number and types of function arguments;
2. number and types of function results;
3. relations existing intra and inter arguments and results;
4. invariant properties of the system under the action of the function;
5. information loss during processing;
6. symmetries in the domain of arguments or results (i.e., mapping of one subdomain onto another in the domain or range of the function).

These properties can also be applied to classification of the 'contents' parts of data structures.

I have been developing over the last year a language called D.D.P.S. (Data Directed Programming System) (Vigor 1967) whose features depend on the concepts briefly described in the first part of this paper.

The internal representation of objects is, in the present version, mainly in the mapped and extended forms, although input, output and backing store messages are at present in a coded form. (This may prove to be a fundamental limitation to the usefulness of the system, as it may obscure the

fundamental simplicity of results requested or obtained from the system.) The objects with which I can operate are the nodes (called switches) in a multifurcating multirelation directed ordered graph, with labelling on both the relations and the nodes. The representation chosen for each node is a map of its structure in the form of:

1. a counter giving its external size in the machine;
2. the address of the indexing operation or 'structure'; and
3. the base address of the contents in its present abode (core store or backing store).

Associated with all nodes which are externally available:

1. there is a name which is the first property and is a character string in coded form;
2. on any node is a 16-valued marker called an item modifier which can be used for tagging;
3. there is a list of addresses of selector functions which allows the retrieval of the next item in any given relation class;
4. there is a node marker, which can be on any switch, which specifies if this switch is to be used in *procedure* or *data* mode. In procedure mode the contents of the active part of a switch are copied, i.e., the selection implied on entry is carried out. In data mode the whole switch word is copied (i.e., the structure and contents are transferred with no operation. By changing this marker from 'data' to 'procedure', we can select, on our next entry into the data structure, the particular sub-structure which is 'relevant' to our particular problem situation. This gives us a method for constructing a program assembler.

The external representation is completely flexible. In it new facilities can be added at will by a 'macro' facility, which allows one to use all the power for dynamic data definition inside each block of the language.

Apart from the general representation discussed above, there are available for specific commonly used types of objects a complete set of constructors and selectors. These carry out updating, compounding, pruning and retrieval of sub-items in the 'natural' representation of this object type. For example (Fig. 1), a 2-array can be represented as

(*a*) A tree with the same number of items on each branch.
(*b*) For matrix operation it may be in the mapped form with a pair of index registers to index it.
(*c*) For output it may be stored in the coded form as a page layout.

Transfer functions between these different representations are available. These are examples of designators and in most cases are one-one and on to. The nature of allowable transformations on programs, and the related data changes to keep the semantics invariant, are being studied in order to gain more understanding of the problem of equivalence of programs, and also to get better criteria for choice of optimal representations.

When setting up a problem on the computer in this way, the initial input would be a functional definition of the problem in terms of subfunctions and names of data items (which one need not have previously defined). Each of these is assigned to an empty 'switch' in which relations are gradually built up. A tree of allowable 'machine features' for the particular machine is used to generate questions, to the user on an on-line teletype, concerning the relations between objects in his problem and the domain of validity of the functions he is defining.

The relations holding over the set of constructors and selectors of the data structure or part thereof, are successively mapped on to the relations holding between machine features. When a relation fits there is a test to see if the machine features concerned can map this particular section of the data structure.

The cost C of this 'fit' can be given by an expression of the form

$$C=(t_T+t_O)P_F$$

where t_T = time taken to transfer the data at this stage of the program into the required form and location for the machine feature to act on it;

t_O = time associated with operations expected to be carried out on this section of the data structure;

P_F = machine feature price.

A crude total 'cost' for the job is obtained and a corresponding program is chosen as a specification of the representation. The selector and constructor functions for the chosen machine features are substituted into the 'function' or 'macro' definition table with pointers from the corresponding external names of these constructors and selectors in the problem specification.

When a program to carry out some operation on the structure is assembled, this constructor and selector specification is made available by changing a switch marker from data to procedure node, on that branch which is selecting the next function to be substituted, in the program stream, in place of the external names.

The simplest cases one can consider as relational properties are the properties of (1) associativity and (2) commutativity of function arguments. (1) can be mapped on to the property of sequential indexing in a real computer. (2) can be shown by storing all the characters at the same hierarchical level in store.

A more striking example is the mapping of symmetrically related objects on to the same area of store, in those operations for which symmetry results in equivalent processes being carried out on the symmetrically related 'names'. The 'ordinal' property of the cardinal numbers denoting set size can be used as criteria for ordering sets in an 'optimal manner'.

The above methodology has been hand tested in two problems, Tic Tac Toe (Noughts and Crosses) and Checkers (Draughts). From a very simple specification of the problems, fairly efficient internal representations were

obtained. No playing strategy has been considered and the tests are still at a fairly rudimentary stage. The results are highly encouraging. A complete checkers board situation can be represented in three words of 48 bits with very simple move and capture generation for both Queens and Pawns, and a trivial Queening operation.

The increase in the capability of a programmer, if he is freed from the burden of choosing a detailed representation for his problem, cannot as yet be fully assessed. The vast leaps made in the complexity of feasible programs on the introduction of ALGOL and FORTRAN are indicative of the advances likely to come from such a step.

The objects which we have been storing and manipulating, and the functions which manipulate them, are simple 'concepts' according to the definition at the start of this paper.

The transformations which can be carried out on programs by changing the representation of the data and the accessing functions are capable of generalisation. By systematic substitution of simple subfunctions for others of the same nature (same number and types of arguments) we can generate analogous programs (Popplestone, pp. 185-94 of this volume, for an example in finite field arithmetic). A method of concept formation, closely related to Foster's syntactic generalisations (J. M. Foster, pp. 195-203 of this volume) is that obtained by destroying certain invariances or establishing new ones inside the system, and choosing functions to conform with these. We obtain, by a combination of these methods, a type of semantic generalisation which we might come to think resembled human conceptualisation.

REFERENCES

Church, A. (1941). The calculi of lambda-conversion. *Annals of Mathematical Studies*, No. 6. Princeton N.J.: Princeton University Press.

Dennis, J. B. (1965). Segmentation and the design of multiprogrammed computer systems. *J. Assn Comput. Mach.*, **12**, 589-602.

English Electric Leo Marconi (1965). KDF9 ALGOL User's Manual, *E.E.L.M. Document.*

Hunt, Earl B. (1962). *Concept Learning—An Information Processing Problem.* New York and London: Wiley.

I.B.M. Corp. (1965). 'PL1' Systems Reference Library File No. 5360-29, Form (28-6571-0).

Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer J.*, **6**, 308-320.

Landin, P. J. (1965). A correspondence between ALGOL-60 and Church's Lamb da-notation. *Communs Assn Comput. Mach.*, **8**, 89-101, 158-165.

Landin, P. J. (1966). The next 100 programming languages. *Communs Assn Comput. Mach.*, **9**, 157-166.

McCarthy, J. (Ed.), Abrahams, P. W., Edwards, D. J., Hart, T. P., & Levin, M. I. (1962). *LISP 1·5 Programmer's Manual.* M.I.T. Press.

Newell, A. (Ed.), Baker, C. L., Feigenbaum, E. A., Green, B. F., Jr., Kelly, H. S., Mealy, G. H., Saber, N., Tonge, F. N., Van Wermer, T., & Wolf, A. K. (1961). '*IPL5 Programming Manual*'. Rand Corporation, Santa Monica, California.

Patterson, J., & Vigor, D. (1967). *On the Inversion and Optimisation of Usercode Programs*, Glasgow Univ. Computing Dept. Report No. 2.

Roberts, L. G. (1965). 'Graphical Communication and Control Languages.' A Lincoln Lab. M.I.T. Report.

Rutledge, J. D. (1964). Ianov schemata. *J. Assn Comput. Mach.*, **11**, 1.

Scoins, I. (1967). Linear graphs and trees. *Machine Intelligence* 1. N. L. Collins and D. Michie (eds.), pp. 3-15. Edinburgh: Oliver and Boyd.

Smith, D. L. (1966). Models and data structures for digital simulation. *Project MAC-TR*-31, M.I.T. Report.

Strachey, C. (1965). A general purpose macro generator; *Brit. Computer Journal*, **8**, 225-241.

Vigor, D. (1967). *D.D.P.S. Programming Manual*, Glasgow Univ. Computing Dept. Report No. 3.

MECHANISED MATHEMATICS

4

AN APPROACH TO ANALYTIC INTEGRATION USING ORDERED ALGEBRAIC EXPRESSIONS

L. I. HODGSON
COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF MANCHESTER

INTRODUCTION

This paper describes an attempt to develop a system of mathematical analysis which is suitable for use by a computer. The ideas are applicable to many variables but in order to 'get the feel' the analysis is restricted to one variable. Together with this mathematical approach the techniques which are used to implement these methods on the computer are discussed. These computing techniques, however, are straightforward and require no further explanation until they are discussed in detail (pp. 53-56).

In order to develop a system of analysis with particular reference to computer use it is necessary to examine the existing analysis to determine why this is not fully suitable for computer use. Several operations in conventional analysis can be performed by a computer in a fairly straightforward manner. In order to perform these operations except at a trivial level it is necessary to use the concept of 'function' which is not clearly defined but is taken to be what the average mathematician means by a function, e.g.,

$$\begin{aligned} & \cos(x) \\ \text{or} \quad & \exp(x) \\ \text{or} \quad & p(x) \end{aligned}$$

One operation which has been treated by such computer techniques is differentiation. These treatments usually involve storing within the computer the rules for differentiation and the properties of the various functions that may arise. Obviously the derivative of any function, or any expression

involving that function, whose properties are not stored cannot be evaluated, thus making the treatment restrictive. The stored properties of the functions needed for differentiation have been developed by generations of mathematicians using more fundamental properties of the said functions and the rules for differentiation. It would seem to be a reasonable approach to develop an analysis in which a definition of a function enabled its derivative to be evaluated in a mechanical manner. This is the approach used in the analysis developed.

Indefinite integration is more difficult than differentiation and the cleverest mathematician could be set a problem by a novice which the expert could not solve and which he couldn't show that he couldn't solve. It would therefore seem unreasonable to expect a mere computer to deal with the problem of integration as usually posed. So a different approach to the concept of integration is presented which is explained in detail later.

MATHEMATICAL ANALYSIS

This analysis involves constants from the field of numbers, an indeterminate x, positive integers and functions of x. The first three of these can be used to form quantities termed subunits defined as follows

$$\text{const. } x^{(\text{power})}$$

with specific examples of subunits

$$2x^4,\ -75{\cdot}27x^{29} \text{ and } (2-i)x^{71}.$$

By additions of subunits a string ξ can be formed

$$\text{e.g., } \xi = 2x^4 - 75{\cdot}27x^{29} + (2-i)x^{71}.$$

Differentiation of a string ξ formed of such subunits is the sum of the derivatives of the subunits.

For example, if $\xi = a_1x^{n1} + a_2x^{n2} + \ldots ar\, x^{nr}$

then $$\xi' = n_1.a_1x^{n1-1} + n_2a_2x^{n2-1} + \ldots nr.ar\, x^{nr-1}.$$

The derivative of a constant is zero.

The functions mentioned earlier can now be defined by using strings and subunits.

For example, $\psi'(x) = \psi(x)$ defines within an arbitrary constant the function $\psi(x)$.

A function ψ can be defined by the following general method. The function ψ has a derivative ψ' and this derivative is set equal to a string which has subunits consisting of some or all of the following elements:

(i) a variable x,
(ii) the function,
(iii) other functions defined earlier by a similar technique.

Examples of these functions are

$$\psi_1(x),\ \psi_2(x) \text{ and } \psi_3(x)$$

where

$$\psi_1'(x)=\psi_1(x),$$
$$\psi_2'(x)=\psi_1(x^2),$$
$$\psi_3'(x)=\psi_1(x)+\psi_3(x).$$

Strings involving these functions can be generated for example ξ_1 and ξ_2 where

$$\xi_1=ax^5\psi_1(x)-7\cdot 2x^{27}\psi_2(x),$$
$$\xi_2=(a+ib)x^7\psi_2(x)+7x^{29}\psi_1(x).$$

A more detailed examination of integration can now be presented. From a naive point of view, integration is just the reverse of differentiation. However, all that is needed from integration is an algebraic string ξ which can be used in exactly the manner as strings ξ_1 or ξ_2 defined earlier. Many such strings can be chosen to satisfy the integral, but it is obviously desirable to choose the simplest of these strings.

Therefore if the integral is not expressible as a simpler combination of simpler functions than itself then choose a new function, whose derivative is equal to the integrand, as the value of the integral.

For example, if $\psi_1'(x)=\psi_1(x)$ and $\int\psi_1(x^2)dx$ is required, then if no simpler combination of simpler functions than $\psi_1(x^2)$ satisfy the integral, then define a new function $\psi_2(x)$ where $\psi_2'(x)=\psi_1(x^2)$.

In order to make any progress with this idea of integration a more precise definition of the meaning of simple is needed. Any definition will do provided it is consistent, e.g.,

$$\text{if } x_1>x_2 \text{ and } x_2>x_3, \text{ then } x_1>x_3.$$

The method used involves the following parameters:

(i) length of string, i.e., number of subunits;
(ii) multiplicity of subunits, i.e., number of functions in the subunit;
(iii) depth of the string, i.e., the maximum depth of the function tree.

To illustrate the ideas consider 2 strings ξ_1 and ξ_2. Let m_1 be the maximum of the length, multiplicity and depth of ξ_1 and m_2 be the corresponding quantity for ξ_2. If $m_1>m_2$ then $\xi_1>\xi_2$. If $m_1=m_2$ then $\xi_1>\xi_2$ if the depth of $\xi_1>$depth of ξ_2. If, however, the depth is the same, then test the multiplicity of the two strings and if there is equality then test the corresponding lengths. If none of these tests is conclusive, then compare the corresponding substrings. These substrings are arranged in order of decreasing complexity.

$\xi_1=\xi_2$ if none of these tests is conclusive.

If the approach to integration is to be any use then there must be a method to test whether a selected string is equal to the integral. This technique is demonstrated by considering the following problem.

Is
$$ax^n\psi_2=\int bx^m\psi_1 dx \qquad (1)$$
with
$$\psi_1'(x)=px^q\psi_1(x)$$
$$\psi_2'(x)=rx^s\psi_2(x)$$

and $n>0$, $m>0$, $p>0$, $q>0$?

Now if (1) is true, then its derivative is also true, i.e.,

$$nax^{n-1}\psi_2+arx^{n+s}\psi_2=bx^m\psi_1$$

or
$$(nax^{n-1}+arx^{n+s})\psi_2=bx^m\psi_1 \tag{2}$$

Differentiating (2) gives

$$[n(n-1)ax^{n-2}+ar(n+s)x^{n+s-1}+nra\,x^{n+s-1}+ar^2x^{n+2s}]\psi_2$$
$$=[mb\,x^{m-1}+bp\,x^{m+q}]\psi_1.$$

On eliminating ψ_1 and ψ_2 assuming neither are zero, the result is

$$bx^m[n(n-1)ax^{n-2}+ar(n+s)x^{n+s-1}+nar\,x^{n+s-1}$$
$$+ar^2x^{n+2s}]=[mb\,x^{m-1}+bp\,x^{m+q}][na\,x^{n-1}+ar\,x^{n+s}].$$

Re-arrangement gives

$$abx^{m+n-2}[mn+mr\,x^{s+1}+pn\,x^{q+1}+pr\,x^{q+s+2}]$$
$$=abx^{m+n-2}[n(n-1)+r(n+s)x^{s+1}+nr\,x^{s+1}+r^2x^{2s+2}] \tag{3}$$

This relation must be true for all values of x for (1) to be true and so the factor $ab\,x^{m+n-2}$ can be cancelled. None of the powers of x can be zero as this would contradict the assumptions concerning q, s.

Consequently $n(n-1)=m.n$, with the result that $n=0$ or $m=n-1$. $n=0$ contradicts the initial assumptions, therefore $m=n-1$. With this value for m equation (3) reduces to

$$(n-1)rx^{s+1}+pn\,x^{q+1}+pr\,x^{q+s+2}$$
$$=(2nr+rs)x^{s+1}+r^2\,x^{2s+2} \tag{4}$$

For this relation to be true,

either $2s+2=q+s+2$

or $2s+2=q+1$

or $s+1=q+s+2$

If $2s+2=q+1$, then the coefficient of x^{q+s+2} must be zero, which gives $pr=0$ contradicting the assumptions.

If $s+1=q+s+2$, then $q=-1$, which contradicts the initial assumptions. Therefore $2s+2=q+s+2$, which gives $s=q$.

Equation (4) reduces to

$$[(n-1)r+pn]x^{s+1}+pr\,x^{2s+2}$$
$$=(2nr+rs)x^{s+1}+r^2\,x^{2s+2} \tag{5}$$

giving $pr=r^2$, which is $p=r$ or $r=0$. The latter contradicts the initial assumptions and so $p=r$ is taken. The other condition from (5) is

$$(n-1)r+rn=2nr+rs$$

which gives $2n-1=2n+s$, i.e.,

$$s=-1$$

which contradicts the initial assumptions. Hence the integral is not equal to the expression (1).

The value $n=0$ although not permissible in the context of the question suggests the relation

$$a\psi_2=\int bx^m\psi_1 dx$$

with $$\psi_1'(x)=px^q\psi_1$$
$$\psi_2'(x)=rx^s\psi_2$$
and $m>0$, $q>0$, $s>0$.

Following a similar technique to that used earlier gives

$$ar\,x^s\psi_2=bx^m\psi_1$$

and $$bx^m(ars\,x^{s-1}+ar^2\,x^{2s})=ar\,x^s(mb\,x^{m-1}+bp\,x^{m+2})$$

or $$abr\,x^{m+s-1}(s+r\,x^{s+1})=abr\,x^{m+s-1}(m+px^{q+1})$$

i.e., $$s+r\,x^{s+1}=m+p\,x^{q+1}$$

giving $m=s$, $q=s$ and $p=r$, with a and r undetermined.

These two examples are carried out using expressions linear in the ψ and so represent a special case but give a good overall idea of the methods used. Further progress requires more precise specification of the type of functions and strings that are used. These strings fall into three categories:

(i) linear in the functions;
(ii) non-linear in the functions but algebraic;
(iii) non-algebraic in the functions.

Category (i) is solvable using the techniques mentioned earlier and is to be investigated by means of a computer.

Category (ii) is solvable but the techniques are not general enough to use computer programs.

Category (iii) is not known to be solvable although there is no reason to suspect that it is not.

The structure of category (i) is arranged in ascended order of complexity for illustration.

$$\psi_1'=ax^n$$
$$\psi_1'=ax^n\psi_1$$
$$\psi_2'=ax^n+bx^m \qquad m\neq n$$
$$\psi_2'=bx^m\psi_1$$
$$\psi_2'=ax^n\psi_2+bx^m$$
$$\psi_2'=ax^n\psi_2+bx^m\psi_1$$
$$\psi_3'=ax^n+bx^m+cx^p \qquad m\neq n\neq p$$
$$\psi_3'=cx^p\psi_2$$
$$\psi_3'=bx^n+cx^p\psi_2$$
$$\psi_3'=ax^n+bx^m\psi_2+cx^p\psi_1\ldots$$
$$\psi_4'=ax^n+hx^m+cx^p+dx^q \qquad n\neq m\neq p\neq q\ldots$$

Integration as defined earlier consists of determining which of these strings which do not have the same ψ on both sides of the expression are related to simpler functions.

Much more investigation needs to be carried out on categories (ii) and (iii), which are of greater interest. Category (ii) corresponds roughly to the functions used in conventional analysis although it is designed to generate any functions required, as indeed are all the categories.

The existence of the function ψ is examined by means of 'connected regions'. The technique is illustrated by considering the class of functions defined by

$$\psi'(x)=\psi(x).$$

Take the point M (>0) in ψ space. A near point $M_1(=M+M\delta x)$ can be made to approach M as $|\delta x|\to 0$ but is distinct from M if $|\delta x|\neq 0$. Similar points M_2, M_3, . . . and M_n can be developed where $M_n=M(1+\delta x)^n$. If $|\delta x|\neq 0$ then $M\neq M_1\neq M_2\neq \ldots \neq M_n$ but the points will approach equality as $|\delta x|\to 0$. Therefore starting from M a region of ψ space can be constructed every point of which can be reached from M. However if $M=0$ then whatever the value of $|\delta x|$, provided that it is bounded, will always give $M_1=M_2=\ldots=M_n=M$. Points with this property are termed 'lonely points'. As the direction in a connected region can be reversed that is if $M_1\to M_2$ then $M_2\to M_1$, then it follows that as a lonely point cannot be left then it cannot be reached. Hence the name 'lonely point'.

A function ψ may have more than one connected region, but these must be separated by lonely points. A function ψ which consists only of lonely points is said to be non-existent.

Any function ψ defined by the method given on page 48 is determined by an arbitrary constant, i.e., if $\psi(x)$ satisfies $\psi'(x)=\psi(x)$ then so does $\psi(x+c)$. For use in function strings and trees in general the function ψ must be set to a specific value although the particular value chosen is usually immaterial. Choosing the arbitrary constant is equivalent to relating the connected regions of ψ space to x space. Therefore the value of the particular constant must give the value of the function inside a connected region.

Consider for example $\psi_1'(x)=-\psi_1^2(x)$ and let the value $\psi_1=1$ be related to $x=0$, i.e., $\psi_1(0)=1$. One of the properties of the class of functions ψ_1 is that $x\psi_1(x)+c\psi_1(x)=1$ and substitution of $\psi_1(0)=1$ in this gives $c=1$. Therefore the particular ψ_1 chosen from the general class $\psi_1'(x)=-\psi_1^2(x)$ satisfies

$$x\psi_1(x)+\psi_1(x)=1$$

This relation is equivalent to

$$x.\frac{1}{x}=1$$

of conventional analysis, or more precisely

$$(x+1).(x+1)^{-1}=1.$$

To further illustrate these ideas, consider the functions

$$\begin{aligned}&\psi_1'(x)=\psi_1(x), && \psi_1(0)=1\\ &\psi_2'(x)=-\psi_2^2(x), && \psi_2(0)=1\\ &\psi_3'(x)=\psi_2(x), && \psi_3(0)=1\end{aligned}$$

which give rise to the string

$$\psi_1(\psi_3(x))=x+1$$

or on re-writing

$$k(x)=x+1$$

where $k'(x)=k(x).\psi_2(x)$.

Now, differentiation of $k(x)=x+1$ gives

$$k(x)\psi_2(x)=1$$

and substitution for $k(x)$ yields

$$x\psi_2(x)+\psi_2(x)=1$$

which is the inverting relationship mentioned earlier. The relationship $k(x)=x+1$ is equivalent to the conventional relationship $\exp(\log_e x)=x$ and the analysis above shows this to be equivalent to

$$.x.\frac{1}{x}=1.$$

The analysis above also indicates how to use functions precisely defined to prove relationships. An interesting point is that the function ψ_3 is defined by an integral relationship, i.e.,

$$\psi_3=\int\psi_2(x)dx$$

but is used in a straightforward manner in developing strings. Other such relationships and functions have been studied, but are omitted in order not to cloud the main principles with detail. One of the objects of the author in carrying out this research was to investigate such relations on a computer.

PROGRAMS WRITTEN TO EXPLORE THE IDEAS

The system of programs developed to investigate the ideas discussed earlier was designed to allow flexibility. This was achieved by writing the program in several phases. The first phase was a language of a similar nature to the compiler compiler of R. A. Brooker *et al.* (Rosen 1964) but very much simpler. The input for this program was: (i) phrase definitions; (ii) formats; (iii) format routines; (iv) machine orders used by format routines; (v) a few alphanumeric lines, e.g., NEWLINE & END.

When these categories had been input then the phase 1 program entered phase 2, which consisted of reading one line of input and then comparing with the format list until recognition was achieved. If recognition had not been achieved when the format list was exhausted, then phase 2 program printed a suitable caption and also set a fault indicator.

The following list of phrases and formats defines the function strings manipulated by the system:

PHRASE [1] =0, 1, 2, 3, 4, 5, 6, 7, 8, 9
PHRASE [2] =[1] [2], [1]
PHRASE [3] =[2].[2], .[2], [2]., [2]
PHRASE [4] =*A*[2],+[3],−[3],−
PHRASE [5] =[0][2]:[19][0], [0][2]:[0], [0]:[19][0]
PHRASE [6] =ψ[5] ($\downarrow$ [2])
PHRASE [7] =[6][7], [6]
PHRASE [8] =[4]*X*[5][7], [4][7], [4]*X*5], [4]
PHRASE [9] =[8][9], [8]

PHRASE [10]=[0], >, **c**
PHRASE [11]=<∅[2], ↓[2], ψ'[2]

PHRASE [19]=[2], N[2]
PHRASE [20]=+, −
PHRASE [21]=A[2][21], A[2]
PHRASE [22]=[20][21][22], [20][21]
PHRASE [23]=N[2][23], N[2]
PHRASE [24]=[20][23][24], [20][23]

The corresponding format line which uses these phrases is

FORMAT [13]=[11]=[9][10]

which defines a general type of function tree satisfying any of the three categories of page 51.

Further explanation of the phrase list is necessary to clarify it. The number between square brackets after the word PHRASE is the phrase number and identifies that phrase. Whenever a number between square brackets occurs on the right-hand side of a phrase definition then that number refers to the phrase of that number. Alternative phrases on the right-hand side of a phrase definition are separated by the symbol ','. Finally the system has a built in definition

PHRASE [0]=[,], ,

which allows these symbols to be part of a phrase definition.

A more detailed examination of

FORMAT [13]=[11]=[9][10]

shows that [11] defines the type of string as follows:

<∅[2] indicates that the string of number [2] has just been started;

[2] indicates that the string number [2] is an argument of some function;

[2] indicates that the string defines a function [2] needed in the overall function tree.

PHRASE [10] defines the end of the string if the category of [10] is [0] then the function tree is not complete but the current string is. If the category is > then both the function tree and the current string are complete.

Finally if the category is *c* then the string is not complete and will be finished on the next line.

PHRASE [9] defines the right-hand side of the string with the exception of the terminator [10]. As can be seen from the definition of [9] it consists of a series of subunits which are defined by [8].

PHRASE [8] can be of four possible types, but the most general is [4]X[5][7] and is of the form: const. variable power.functions.

PHRASE [5] defines a type of variable or function by two numbers, one of which is the variable or function number and the other is the power of the variable or function.

PHRASE [7] defines the product of functions or a single function.

PHRASE [6] is the single function ψ followed by [5] and finally by an argument $\downarrow$ [2].

A simple example of a function corresponding to FORMAT [13] is

$$\begin{aligned}&\langle \emptyset 1 = +A1x[1:2]\psi[1:2](\downarrow 1)+A2x[1:1],\\ &A2 = +A2,\\ &A1 = +A1,\\ &\downarrow 1 = +79{\cdot}2x[1:N1],\\ &N1 = +N1,\\ &\psi'1 = +A1\psi[1:1](\downarrow 1)\rangle\end{aligned}$$

All these phrase and format definitions define the 'form' of the function string or tree, but provide no facilities for changing the function tree in a manner similar to that in conventional languages which allow variables to be charged dynamically. Nor do they allow decisions to be taken depending on the state of the function tree. The following list of phrases and formats allow such facilities.

PHRASE [25] = [2]:$\emptyset$[2]:, $\emptyset$[2]:
PHRASE [26] = $\emptyset$[2], $\downarrow$ [2],ψ^1[2]
PHRASE [27] = N[2], A[2]
PHRASE [28] = i, j, k, l, m, n
PHRASE [29] = [2], $-$[2], [28]
PHRASE [30] = $\emptyset$[29], [29], [29]
PHRASE [31] = $L\emptyset$[29][30], $M\emptyset$[29][30]([39])
PHRASE [32] = $>$, $>$, $=$, $<$, $<$, $\neq$
PHRASE [33] = [31], [29]
PHRASE [34] = +, $-$
PHRASE [35] = [2]: $->$, $->$
PHRASE [36] = [2]: [28], [28]

FORMAT [23] = $\emptyset$[2]: =$\emptyset$[2]
FORMAT [24] = [2]: $\emptyset$[2]=$\emptyset$[2]
FORMAT [25] = [25][26]: =[9] [10]
FORMAT [26] = [25][27]: =[27][10]
FORMAT [27] = [25] [26]: =[26][9][10]
FORMAT [28] = [35] [2] **if** [33] [32] [33]
FORMAT [29] = [35][2]
FORMAT [30] = [36]=[29] [34] [29]
FORMAT [31] = [36]=[33]

Formats 23-27 allow function trees to be changed dynamically.

FORMAT [28] allows conditional decisions to be taken. For example

$->3$ **if** $L\emptyset 3\psi'1<4$
or $->3$ **if** $L\emptyset 3\emptyset 3<4$
or $->3$ **if** $M\emptyset 3\psi'\ 1(4)\neq 7$

The indicator *L* denotes the length of the string which is defined by the two string names immediately adjacent to the *L*.

The letter *M* indicates the multiplicity of the subunit in the string defined by the two string names. The particular subunit is determined by the quantity in the brackets. By the use of these conditional instructions and the dynamic alteration of function trees the programming system becomes much more powerful.

The immediate object of the programming system is to investigate more fully categories (i) and (ii). The use of a computer enables the investigation to be carried out much more rapidly and may help to suggest ideas to solve many remaining problems of which the following are examples:

(i) how to deal with category (iii);
(ii) how to determine the true properties of defined functions;
(iii) to investigate the solution of differential equations;
(iv) study *n*-dimensional space.

REFERENCE

Rosen, Saul (1964). A compiler building system developed by Brooker and Morris. *Communs Assn Comput. Mach.*, **7**, 403-414.

5

SOME THEOREM-PROVING STRATEGIES BASED ON THE RESOLUTION PRINCIPLE

JARED L. DARLINGTON
RHEINISCH-WESTFÄLISCHES INSTITUT
FÜR INSTRUMENTELLE MATHEMATIK
BONN, GERMANY

The formulation of the resolution principle by J. A. Robinson (1965a) has provided the impetus for a number of recent efforts in automatic theorem-proving. In particular, the program PG1 (Wos *et al.* 1964, 1965), written by L. Wos, G. A. Robinson and D. F. Carson for the Control Data 3600, utilises the resolution principle in conjunction with a 'unit preference strategy' and a 'set of support strategy' to produce efficient proofs in first-order functional logic and group theory. The present author also has experimented with the resolution principle, and has incorporated it into a pair of COMIT theorem-proving programs written at the Institut für Instrumentelle Mathematik in Bonn and currently running on the Institute's IBM 7090. These programs have generated proofs of some interesting propositions of number theory, in addition to theorems of first-order functional logic and group theory.

The resolution principle, like many other theorem-proving methods, is a refutation algorithm, in the sense that it seeks to generate a contradiction from an initial set of clauses

$$C_1, C_2, \ldots, C_k$$

resulting from the negation of the theorem or formula to be tested, usually in conjunction with some already established axioms or theorems. These initial or input clauses C_i are a set of logical expressions in conjunctive normal form: that is, the C_i are regarded as joined by logical 'and', while an

individual clause is either a single literal L, in which case it is called a 'one-literal clause' or a 'unit clause', or a disjunction of n literals

$$L_1 \vee L_2 \vee \ldots \vee L_n$$

which case it is called an 'n-literal clause' or simply an 'n-clause'. A 'literal' is an n-place predicate expression

$$F(x_1, x_2, \ldots, x_n)$$

or its negation

$$\bar{F}(x_1, x_2, \ldots, x_n)$$

whose arguments are individual variables, individual constants, or functional expressions. Quantifiers do not occur in these formulae, since existentially quantified variables have been replaced by functions of universally quantified ones, and the remaining variables may therefore be taken as universally quantified. For example, the number-theoretic proposition

> 'For all x and y, if x is a divisor of y then there exists some z such that x times z equals y'

may be symbolised as

(i) $$\bar{D}(x, y) \vee T(x, f(x, y), y)$$

in which '$D(x, y)$' stands for 'x is a divisor of y' and '$T(x, y, z)$' stands for 'x times y equals z'. The formula (i) is a 2-literal clause, and may be read as follows:

> 'Either x is not a divisor of y, or there is a function $f(x, y)$ of x and y such that x times $f(x, y)$ equals y'.

Another example is the 'first proposition of Euclid':

> 'For all x, y, and z, if x is a prime and x is a divisor of $y.z$, then x is a divisor of either y or z',

which may be symbolised as

(ii) $$\bar{P}(x) \vee \bar{T}(y, z, u) \vee \bar{D}(x, u) \vee D(x, y) \vee D(x, z).$$

Both (i) and (ii) are general formulae, which admit of infinitely many instantiations, or substitution instances, within the positive integers. Exhaustive instantiation, in fact, was the basis of a number of early theorem-proving programs. Starting with a finite number of individual constants, all possible substitutions were made in the input clauses, and the resulting conjunction of substitution instances was tested for truth-functional consistency. If a contradiction was not obtained, more constants were produced by substituting the available constants for the variables in the functional expressions $f(x, y)$, etc., that stood for the existential variables, and the additional constants were used to generate more substitution instances. This process continued until a contradiction was obtained or (the more usual case) the machine ran out of time or storage. The Herbrand theorem guaranteed that exhaustive instantiation would eventually produce a proof if one existed, given enough time and storage, but the large number of irrelevant substitution instances

generated often prevented the machine finding proofs of even relatively simple theorems. It soon became evident, therefore, to researchers in the field that the primary problem in automatic theorem-proving is that of impeding the generation of irrelevant clauses and inferences. A way in which this might be done was suggested by D. Prawitz *et al.* (1960) and elaborated upon by M. Davis (1963). The basic idea was that any substitution instance of a clause is bound to be irrelevant to a proof so long as the literals occurring in it are not 'mated', i.e., negated, by other literals occurring in other clauses. Davis in fact proved that any substitution instance

$$L_1 \vee L_2 \vee \ldots \vee L_n$$

that contains at least one unmated literal may be erased without affecting consistency, and that the test for consistency may be confined to 'linked conjuncts', i.e., conjuncts (or clauses) wherein each literal L_i is negated by a mate $\bar{L}_i$ occurring in some other clause. This 'theorem on linked conjuncts' provides a necessary, though not a sufficient, condition of relevance: any substitution instance containing an unmated literal is demonstrably irrelevant to consistency and may be deleted forthwith, but the remaining substitution instances are not necessarily all relevant. Davis showed (1963) that a theorem-proving strategy based on the search for linked conjuncts is capable of producing short proofs of many theorems, and gave as a detailed example the proposition that any left inverse in an associative group is also a right inverse.

The resolution principle takes over the idea of searching for mated literals, but applies it directly to the original clauses rather than to their substitution instances. Instead of searching for mates among the substitution instances, the resolution principle employs an algorithmic procedure to determine in advance whether two literals L_1 and L_2 in two different clauses could yield contradictory substitution instances. For example, if L_1 and L_2 are

(iii) $$T(G(x, y), x, y)$$

and

(iv) $$\bar{T}(G(u, K(v)), w, K(v)),$$

respectively, it may be determined that these two clauses will yield contradictory substitution instances, and that these (apart from the negation sign in (iv)) are all instances of the formula

(v) $$T(G(x, K(y)), x, K(y)).$$

In the terminology of J. R. Guard (1964), formula (v) may be called a 'general matching formula' ('GMF') for L_1 and $\bar{L}_2$, since it stands for all the common substitution instances of L_1 and $\bar{L}_2$. Whenever a pair of clauses C_1 and C_2 can be found that contain literals L_1 and L_2 such that L_1 matches $\bar{L}_2$, i.e., such that L_1 and $\bar{L}_2$ have common substitution instances, then a 'resolvent' of C_1 and C_2 may be generated by disjunctively joining C_1' and C_2', where C_1' is formed by making the same substitutions throughout $C_1 - L_1$ that are necessary to make L_1 equal to the GMF, and C_2' is formed by making the same substitutions throughout $C_2 - L_2$ that are necessary to make $\bar{L}_2$ equal

to the GMF. For example, suppose (iv) is replaced by the 2-literal clause

(vi) $$\bar{T}(G(u, K(v)), w, K(v)) \vee T(u, v, w)$$

in which the first literal is identical to (iv). Then (v) is the GMF of (iii) and the negation of the first literal in (vi), and the latter will equal the GMF (v) so long as the following substitutions are made throughout (vi):

$$u=x, v=y, w=x$$

This leaves

(vii) $$T(x, y, x)$$

as the resolvent of (iii) and (vi). Resolution is thus a functional logic analogue of the operation of 'cut' in propositional logic, whereby a single clause

$$K_2 \vee K_3 \vee \ldots \vee K_m \vee L_2 \vee L_3 \vee \ldots \vee L_n$$

can be deduced from the two separate clauses

$$K_1 \vee K_2 \vee \ldots \vee K_m$$

and

$$\bar{K}_1 \vee L_2 \vee \ldots \vee L_n.$$

If C_1 and C_2 are both unit clauses, as in (iii) and (iv), then there is nothing left over after the resolvent is formed: that is, the sole resolvent of two contradictory unit clauses is the null clause. Such a result amounts to a proof that the initial set C of clauses is unsatisfiable. The resolution principle searches for a contradiction between two unit clauses, and if none can be found, it generates more resolvents until a pair of contradictory unit clauses is produced. The algorithm in its original form called for the generation of

$$S_0, S_1, S_2, \ldots, S_i, \ldots$$

where S_0 is the set of input clauses, and S_i consists of S_{i-1} plus all resolvents and 'factors' (see below) of resolvents of the elements of S_{i-1}. This procedure, though provably complete (as shown by Robinson in 1965a), is extremely wasteful, and is hardly more efficient than the earlier methods based on exhaustive instantiation. The reason for this inefficiency is that resolution normally produces longer and longer clauses (a resolvent of a 3-clause and a 4-clause, for example, is ordinarily a 5-clause, unless one or more of its literals are redundant), while a proof of unsatisfiability requires the generation of shorter, and ultimately unit, clauses. One way of generating shorter clauses is by means of a technique called 'factoring' (by Robinson), whereby a clause C_i may be factored if it contains two matching literals L_1 and L_2. The factor is formed by making the same substitutions throughout C_i that are necessary to make L_1 equal to L_2, and then deleting one of these two literals along with any other redundant literals. The following clause

(viii) $$\bar{P}(x) \vee \bar{T}(y, y, u) \vee \bar{D}(x, u) \vee D(x, y)$$

is a factor of the earlier example (ii), formed in an obvious way by substituting y for z throughout (ii) and then deleting the redundant literal $D(x, y)$.

A second and more generally applicable technique for generating shorter clauses is to resolve unit clauses against clauses of length $n(n \geq 1)$, thereby

producing clauses of length $n-1$ or less. It turns out in fact that many theorems can be proven by generating only resolvents that have at least one unit clause as parent ('resolvend'), and the resulting simplification of proofs is considerable.

There are several ways of writing a program that gives priority to unit clauses and the resolvents descended from them. In addition to our two COMIT programs, which we may call 'D1' and 'D2', there is the afore-mentioned program PG1 of Wos *et al.* D1 proceeds by generating successively S_0, S_1, S_2, etc., exactly as the original resolution algorithm, but with the restriction that each resolvent must have at least one unit clause as a parent, or (to allow for the possibility that there may initially be no unit clauses) at least one n-clause ($n \geqslant 1$) as a parent, where n is the length of the shortest clause. D2 proceeds by resolving the unit clauses (or the n-clauses, as in D1) first against each other, then against the first 2-clause, the second 2-clause, etc., then against the first 3-clause, the second 3-clause, etc. Whenever two clauses C_1 and C_2 produce one or more resolvents R_1, R_2, etc., then the R_i take priority over the older clauses in the sense that they are stored at the front of their appropriate lists (unit clauses, 2-clauses, etc.) and the last R_i to be generated is the first to be resolved against the unit clauses. PG1 is similar to D2, the main difference being that, while D2 generates all the resolvents R_1, R_2, etc., of a pair of clauses C_1 and C_2 before proceeding further, PG1 generates only the first resolvent R_1 (or the ith resolvent R_i, if R_{i-1} has already been generated) of C_1 and C_2 and then immediately proceeds to resolve R_1 (or R_i) against the unit clauses (or the n-clauses, where n, as in D1 and D2, is the length of the shortest clause).

The three methods described may be compared in terms of how many resolvents they generate at a time. D1 generates all the resolvents (provided that each resolvent has at least one n-clause as a parent) of a set C; D2 generates all the resolvents of a given pair C_1 and C_2 (provided that either C_1 or C_2 is an n-clause); and PG1 generates only one resolvent of C_1 and C_2 at a time. The difference between the three programs may be illustrated in terms of the 'Gilmore problem', i.e., the set of clauses

(1) $F(x, y)$
(2) $G(x, y) \vee \bar{F}(y, P(x, y)) \vee \bar{F}(P(x, y), P(x, y))$
(3) $\bar{F}(y, P(x, y)) \vee \bar{F}(P(x, y), P(x, y)) \vee \bar{G}(x, P(x, y)) \vee \bar{G}(P(x, y), P(x, y))$

first used as an example by P. C. Gilmore (1960). D1 generated the following proof of this example in 19 seconds (in this and subsequent examples, we shall omit parentheses within literals where no ambiguity can result):

D1 proof of Gilmore example (19 *seconds*)

(1) Fxy	premise
(2) $Gxy \vee \bar{F}yPxy \vee \bar{F}PxyPxy$	premise
(3) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$	premise
(4) $Gxy \vee \bar{F}PxyPxy$	(1) & (2)
(5) $Gxy \vee \bar{F}yPxy$	(1) & (2)

(6) $\bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$	(1) & (3)
(7) $\bar{F}yPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$	(1) & (3)
(8) Gxy	(1) & (4)
(9) Gxy	(1) & (5)
(10) $\bar{G}xPxy \vee \bar{G}PxyPxy$	(1) & (6)
(11) $\bar{G}xPxy \vee \bar{G}PxyPxy$	(1) & (7)
(12) $\bar{G}PxyPxy$	(8) & (10)
.....	
.....	
(22) contradiction	(8) & (12)

The gap between (12) and (22) is because nine extra clauses were generated before the contradiction between (8) and (12) was discovered. D1 in fact will usually generate additional clauses after a contradictory pair has been generated, because it does not test each unit clause immediately upon generation, as does PG1, but waits until its 'generation' is complete. Furthermore, clauses (5), (7), (9) and (11) are irrelevant to the proof, and need not be shown in the final printout. Two of these, (9) and (11), are actually redundant, and may be deleted.

D2 generated the following proof, in 10 seconds:

D2 proof of Gilmore example (10 *seconds*)

(1) Fxy	premise
(2) $Gxy \vee \bar{F}yPxy \vee \bar{F}PxyPxy$	premise
(3) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$	premise
(4) $Gxy \vee \bar{F}PxyPxy$	(1) & (2)
(5) $Gxy \vee \bar{F}yPxy$	(1) & (2)
(6) Gxy	(1) & (5)
(7) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}PxyPxy$	(3) & (6)
(8) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy$	(3) & (6)
(9) $\bar{F}yPxy \vee \bar{F}PxyPxy$	(6) & (8)
(10) $\bar{F}PxyPxy$	(1) & (9)
(11) $\bar{F}yPxy$	(1) & (9)
(12) contradiction	(1) & (11)

Clauses (4), (7) and (10) are superfluous, and may be omitted from the final printout. In addition, an extra Gxy was generated, from (1) and (4), but was immediately deleted.

Finally, we may present the proof that PG1 would produce if it were given the Gilmore example.

PG1 proof of Gilmore example

(1) Fxy	premise
(2) $Gxy \vee \bar{F}yPxy \vee \bar{F}PxyPxy$	premise
(3) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$	premise
(4) $Gxy \vee \bar{F}PxyPxy$	(1) & (2)
(5) Gxy	(1) & (4)

(6)	$Gxy \vee \bar{F}yPxy$	(1) & (2)
(7)	$\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}PxyPxy$	(3) & (5)
(8)	$\bar{F}yPxy \vee \bar{F}PxyPxy$	(5) & (7)
(9)	$\bar{F}PxyPxy$	(1) & (8)
(10)	contradiction	(1) & (9)

Only two superfluous clauses are generated: (6), and an extra Gxy from (1) and (6).

On the basis of the Gilmore example, it would appear that PG1 is the best, and D1 the worst, of the three programs described, with D2 falling somewhere in between but rather nearer to PG1. This judgment is confirmed by many examples, of which we may present one more: the proof of the group-theoretical proposition, which we may call 'RI', that any associative system that has left and right solutions contains a right identity element.

First D1 proof of RI (322 *seconds*)

(1)	$TGxyxy$	existence of left solution
(2)	$TxHxyy$	existence of right solution
(3)	$TxyFxy$	closure
(4)	$\bar{T}KxxKx$	no right identity
(5)	$\bar{T}xyu \vee \bar{T}yzv \vee \bar{T}xvw \vee Tuzw$	associativity
(6)	$\bar{T}xyu \vee \bar{T}yzv \vee \bar{T}uzw \vee Txvw$	
(8)	$\bar{T}xyu \vee \bar{T}yzy \vee Tuzu$	factor of (5)
(20)	$\bar{T}xyz \vee TzHyyz$	(2) & (8)
(33)	$\bar{T}yzKx \vee \bar{T}zxz$	(4) & (8)
(73)	$TyHxxy$	(1) & (20)
(77)	$\bar{T}xyx$	(1) & (33)
(250)	contradiction	(73) & (77)

Second D1 proof of RI (117 *seconds*)

(1)	$TGxyxy$	
(2)	$TxHxyy$	
(3)	$TxyFxy$	
(4)	$\bar{T}KxxKx$	
(5)	$\bar{T}xyu \vee \bar{T}yzv \vee \bar{T}xvw \vee Tuzw$	
(6)	$\bar{T}xyu \vee \bar{T}yzv \vee \bar{T}uzw \vee Txvw$	
(7)	$\bar{T}yzKx \vee \bar{T}zxu \vee \bar{T}yuKx$	(4) & (5)
(9)	$\bar{T}xyz \vee \bar{T}GxKyzKy$	(1) & (7)
(24)	$\bar{T}xyx$	(1) & (9)
(69)	contradiction	(2) & (24)

D2 proof of RI (11 *seconds*)

(1)	$TGxyxy$
(2)	$TxHxyy$
(3)	$TxyFxy$

(4) $\overline{T}KxxKx$	
(5) $\overline{T}xyu \vee \overline{T}yzv \vee \overline{T}xvw \vee Tuzw$	
(6) $\overline{T}xyu \vee \overline{T}yzv \vee \overline{T}uzw \vee Txvw$	
(7) $\overline{T}yzKx \vee \overline{T}zxu \vee \overline{T}yuKx$	(4) & (5)
(10) $\overline{T}GxKyzKy \vee \overline{T}zyx$	(1) & (7)
(11) $\overline{T}xyx$	(1) & (10)
(13) contradiction	(2) & (11)

PG1 produced a proof of RI essentially the same as that generated by D2 (though PG1 runs much faster than D1 or D2, and obtained its proof of RI in 35 milliseconds).

The latter two proofs depicted above employ an additional heuristic, the 'set of support strategy' of Wos *et al.* (1964, 1965), according to which a subset S of the set C of input clauses is singled out as the 'set of support' of C. This means that every resolvent produced must be a descendant of at least one element of S, i.e., no resolvents are formed entirely from elements of $C-S$. The most natural choice of S is the set of clauses that formulate the special assumptions of the proof and the negation of the conclusion. In the proofs just presented, the set of support consists of the single unit clause (4), the assumption that there exists no right identity element.

Another aspect of the above proofs of RI that should be mentioned is that factoring turns out not to be essential, and is omitted from the latter two proofs, though it is included in the first proof of RI. It is often possible to obtain proofs without factoring, and it is advisable to try to do so, since the generation of factors has the tendency to produce clauses that are too specific to be of any use in a proof. There are, however, some examples in which factoring appears to be essential, such as the following, which we may call 'W5', since it is based on the fifth example of H. Wang (1964) in the appendix (Wang attributes the example to Quine).

D1 proof of W5 (15 *seconds*)

(1) $\overline{G}ya \vee \overline{G}yw \vee \overline{G}wy$	premise
(2) $Gya \vee GyPy$	premise
(3) $Gya \vee GPyy$	premise
(4) $\overline{G}ya \vee \overline{G}ay$	factor of (1)
(6) $\overline{G}ya \vee \overline{G}yy$	factor of (1)
(7) $\overline{G}aa$	factor of (6)
(9) $GaPa$	(2) & (7)
(10) $GPaa$	(3) & (7)
(11) $\overline{G}Paa$	(4) & (9)
(20) contradiction	(10) & (11)

In the above proof, factoring plays an essential role in the derivation of the key unit clause (7) in two steps from (1). It is not essential, however, in the derivation of (11), which could have been produced directly by resolution of (1) and (9), instead of first factoring (1) to obtain (4) and then resolving (4) and (9) to obtain (11).

The superiority, implicit in the foregoing examples, of D2 and PG1 over D1 is not always realised in practice, since there are some proofs for which D1's horizontal or across-the-board method of generating clauses is more suitable than the vertical or up-the-ladder procedure of D2 and PG1. One such example is the number-theoretic proof that every integer greater than one has a prime divisor, which was used as an example by S. A. Cook (1965), and which we may call 'C1'.

D1 proof of C1 (51 seconds)

(1) $L1a$	a is greater than 1
(2) $\bar{P}x \vee \bar{D}xa$	a has no prime divisor
(3) $\bar{L}1x \vee \bar{L}xa \vee PKx$	assumption for induction: every x greater than 1 but less than a has a prime divisor
(4) $\bar{L}1x \vee \bar{L}xa \vee DKxx$	
(5) Dxx	reflexivity of 'divisor'
(6) $Px \vee L1Qx$	if x is not a prime, then x has a divisor greater than 1 but less than x
(7) $Px \vee LQxx$	
(8) $Px \vee DQxx$	
(9) $\bar{D}xy \vee \bar{D}yz \vee Dxz$	transitivity of 'divisor'
(14) $\bar{P}a$	(2) & (5)
(15) $L1Qa$	(6) & (14)
(16) $LQaa$	(7) & (14)
(17) $DQaa$	(8) & (14)
(21) $\bar{L}1Qa \vee PKQa$	(3) & (16)
(22) $\bar{L}1Qa \vee DKQaQa$	(4) & (16)
(24) $\bar{D}xQa \vee Dxa$	(9) & (17)
(28) $PKQa$	(15) & (21)
(29) $DKQaQa$	(15) & (22)
(35) $\bar{D}KQaa$	(2) & (28)
(36) $DKQaa$	(24) & (29)
(43) contradiction	(35) & (36)

(Note that clause (1) is not essential to the proof, since it is implicit in clauses (6)-(8).)

Incomplete D2 proof of C1 (116 seconds)

(1)-(9), as in the D1 proof of C1	
(10) $\bar{P}a$	(2) & (5)
(11) $L1Qa$	(6) & (10)
(12) $LQaa$	(7) & (10)
(13) $DQaa$	(8) & (10)
(14) $\bar{P}Qa$	(2) & (13)
(15) $L1QQa$	(6) & (14)
(16) $LQQaQa$	(7) & (14)

(17) $DQQaQa$	(8) & (14)
(18) $\bar{D}Qaz \vee DQQaz$	(9) & (17)
(19) $\bar{D}xQQa \vee DxQa$	(9) & (17)
(20) $DQQaa$	(13) & (18)
(21) $\bar{P}QQa$	(2) & (20)
(22) $L1QQQa$	(6) & (21)
(23) $LQQQaQQa$	(7) & (21)
(24) $DQQQaQQa$	(8) & (21)
(25) $DQQQaQa$	(19) & (24)
(26) $\bar{D}Qaz \vee DQQQaz$	(9) & (25)
(27) $\bar{D}xQQQa \vee DxQa$	(9) & (25)
(28) $DQQQaa$	(13) & (26)
(29) $\bar{P}QQQa$	(2) & (28)
(30) $L1QQQQa$	(6) & (29)
(31) $LQQQQaQQQa$	(7) & (29)
(32) $DQQQQaQQQa$	(8) & (29)
(33) $DQQQQaQa$	(27) & (32)
(34) $\bar{D}Qaz \vee DQQQQaz$	(9) & (33)
(35) $\bar{D}xQQQQa \vee DxQa$	(9) & (33)
(36) $DQQQQaa$	(13) & (34)
(37) $\bar{P}QQQQa$	(2) & (36)

. . . . . .

. . . . . .

Clauses (1)-(4) in the above two examples are taken as the set of support; together they formulate the assumption that there exists some $a>1$ that has no prime divisor, but every $1<x<a$ has a prime divisor. The trouble with the attempted D2 proof of C1 is that it has got itself into a loop, and is unable to form any resolvents from clauses (3) or (4). In order to obtain the proof, it is necessary to resolve (3) and (4) against (11) and (12), thereby producing $PKQa$ and $DKQaQa$, but this requires that unit clauses be resolved against 3-clauses, and there are always 2-clauses that take priority. A simpler example of looping is the tendency of the pair of clauses

$$Px,\ \bar{P}x \vee PFx$$

to produce

$$PFx,\ PFFx,\ PFFFx,\ \text{etc.}$$

ad infinitum. This sort of looping was recognised as a problem by the authors of PG1, who programmed around the difficulty by placing a limit, called a 'level bound', on the number of times the unit clauses can be resolved against clauses of length m before going on to the $m+1$-clauses. This limit is usually set in advance of computation, at around 4 or 5. The 'level' of the input clauses C is 0, the level of a factor of C_i equals the level of C_i, and the level of a resolvent R of C_1 and C_2 is greater by 1 than the maximum of the levels of C_1 and C_2. If the level bound is set at k, then unit clauses whose level

exceeds k, and non-unit clauses whose level exceeds $k-1$, are simply not generated. In the attempted D2 proof of C1, for example, none of the clauses beyond (17) would be generated if k were set at 5, and the program would be forced to back up and generate the correct resolvents. The level bound strategy depends for its effectiveness upon choosing the correct value of k: if k is too small, the program cannot generate a proof, and if k is too large, the resulting irrelevant unit clauses could seriously delay the generation of a contradiction.

The procedure of D2 and PG1 of exhausting the consequences of the shorter clauses before proceeding to the longer ones creates another kind of difficulty, which becomes evident in problems like the proof of the irrationality of the square root of a prime, which proposition we may call 'SRP'.

D1 proof of SRP (437 seconds)

(1) Pp	p is a prime
(2) $TpKbKa$	$p.b^2=a^2$
(3) $\bar{D}xa \vee \bar{D}xb \vee x=1$	a and b are relatively prime
(4) $\neq p1$	p is not equal to 1
(5) $TxxKx$	$x.x=x^2$
(6) $\bar{T}xyz \vee Dxz$	if $x.y=z$, then x is a divisor of z
(7) $\bar{D}xy \vee TxFxyy$	if x is a divisor of y, then $x.Fxy=y$
(8) $\bar{D}xy \vee \bar{D}yz \vee Dxz$	transitivity of 'divisor'
(9) $\bar{T}xyz \vee \bar{T}zuv \vee \bar{T}xwv \vee Tuyw$	special form of cancellation theorem: if $x.y.u=x.w$, then $u.y=w$ (consequent is commuted form of $y.u=w$)
(10) $\bar{P}x \vee \bar{T}yzu \vee \bar{D}xu \vee Dxy \vee Dxz$	'first proposition of Euclid'
(11) $DpKa$	(2) & (6)
(14) $\bar{T}pyz \vee \bar{T}zuKa \vee TuyKb$	(2) & (9)
(15) $\bar{T}yzu \vee \bar{D}pu \vee Dpy \vee Dpz$	(1) & (10)
(17) $\bar{D}pa \vee \bar{D}pb$	(3) & (4)
(32) $\bar{T}pya \vee TayKb$	(5) & (14)
(35) $\bar{D}pKx \vee Dpx$	(5) & (15)
(48) Dpa	(11) & (35)
(57) $\bar{D}pb$	(17) & (48)
(58) $TpFpaa$	(7) & (48)
(60) $\bar{D}az \vee Dpz$	(8) & (48)
(95) $\bar{D}pKb$	(35) & (57)
(97) $TaFpaKb$	(32) & (58)
(145) $\bar{D}aKb$	(60) & (95)
(156) $DaKb$	(6) & (97)
(250) contradiction	(145) & (156)

In the above proof the first three clauses, which formulate the assumption that there exists a prime whose square root is rational, were taken as the set of support. Without the set of support strategy, D1 was unable to generate a proof of SRP within the storage limitations of the machine. Even with the aid of the set of support strategy, D2 was unable to generate a proof of SRP in 30 minutes, since it never got round to generating the essential clause (35), which is a consequence of (1), (5) and (10), and which states that if p is a divisor of x^2 then p is a divisor of x. An efficient proof requires that (35) be generated fairly early on, but D2 will not form any resolvents with (10), which is the longest clause in the set, until it has first exhausted the consequences of (1)-(9). Furthermore, when D2 starts to work on (10) it resolves it against the most recently generated unit clauses, but the generation of (35) requires that (10) be resolved against the original unit clauses. In other words, D2 (and PG1), which attacks the shortest clauses first, will not produce efficient proofs in cases (such as SRP) where efficiency demands that the longest clauses be attacked first, but D1, which attacks the short and the long clauses more or less equally, will occasionally produce moderately efficient proofs in such cases.

There remains to discuss one further type of example, that which requires the deduction of a contradiction from a set of non-unit clauses. Factoring, as in the case of W5, will sometimes suffice to generate the unit clauses essential to the operation of the programs described here. In other cases, it is necessary to produce unit clauses through resolution of non-unit clauses. This is a matter of finding or generating a pair of 2-clauses

$$C_1 \vee C_2,\ C_3 \vee C_4$$

that yield a resolvent

$$C_5 \vee C_5$$

that collapses into a unit clause, C_5, upon deletion of one of the redundant literals as, for example, in the deduction of

$$PFFGy$$

from

$$PFx \vee Px,\ \bar{P}FGy \vee PFFGy.$$

PG1 makes provision for this sort of inference by the inclusion of a 'non-unit section', which takes over if the 'unit section' fails to generate a proof within the stated level bound. D1 and D2 also implicitly contain non-unit sections, in that they give priority to the n-clauses, which are the shortest non-unit clauses in the event that there are no unit clauses, and their descendants. D1 and D2, however, have no provision for passing to a non-unit section in the event that the unit clauses and their resolvents fail to produce a proof within given limits. An example of how a non-unit section can contribute to a proof is the following, which we may call 'W3', since it is a somewhat simplified version of the third example given by H. Wang (1964) in the appendix (Wang attributes the problem to Church).

D1 proof of W3 (271 seconds)

(1)	$FPx \vee Fx$	premise
(2)	$\bar{H}Px \vee Hx$	premise
(3)	$\bar{H}Px \vee Gx$	premise
(4)	$\bar{G}Px \vee Fx$	premise
(5)	$\bar{F}Px \vee GPx \vee Hx$	premise
(6)	$\bar{F}Qx \vee \bar{G}Qx \vee \bar{H}Qx$	premise
(7)	$\bar{F}Px \vee HPx \vee \bar{G}x$	premise
(9)	$\bar{H}PPx \vee Fx$	(3) & (4)
(19)	$\bar{H}PQx \vee \bar{F}Qx \vee \bar{H}Qx$	(3) & (6)
(20)	$Fx \vee \bar{F}Px \vee Hx$	(4) & (5)
(78)	$Fx \vee Hx$	(1) & (20)
(89)	$\bar{H}PQx \vee \bar{F}Qx$	(2) & (19)
(129)	$FPx \vee Hx$	(2) & (78)
(131)	$\bar{H}PQx \vee \bar{H}PPQx$	(9) & (89)
(145)	$HPx \vee \bar{G}x$	(7) & (78)
(213)	$\bar{H}PPQx$	(2) & (131)
(257)	$Hx \vee GPx$	(5) & (129)
(269)	$GPPPQx$	(213) & (257)
(272)	$\bar{H}PPPQx$	(2) & (213)
(273)	$HPPPPQx$	(145) & (269)
(280)	$\bar{H}PPPPQx$	(2) & (272)
(281)	contradiction	(273) & (280)

Upon the generation of the unit clause (213), the value of n was lowered from 2 to 1, and the 'unit section' took over, producing a series of additional unit clauses that quickly resulted in a contradiction. In order to produce the above proof in a reasonable time, it was necessary to employ a special-purpose deletion heuristic, based on the assumption that a resolvent R of C_1 and C_2, that is no shorter than the longer of C_1 and C_2, is of use in a proof if and only if it assists in the generation of shorter clauses. After a reasonable time, therefore, such resolvents R may be deleted. In the above proof of W3, for example, the resolvents R that are no shorter than the longer of their parents are deleted after clauses shorter than R are produced. This entails the deletion of clauses (19) and (20), along with a great mass of other 3-clauses, after the 2-clauses (78) and (89) are generated. This deletion heuristic was tailored specially to fit W3, and there is no guarantee that it would work in very many cases, but some such heuristic is essential if W3 is to be proven in a reasonable time, since the combinatorial possibilities between the various clauses are very great and the amount of garbage produced therefore increases at a rapid rate.

W3 is another example for which the procedure of D1 is somewhat more efficient than that of D2, since D1 resolves the 2-clauses fairly immediately against the 3-clauses, thereby generating the important 3-clauses (19) and (20) at an early stage in the proof, while D2 (and PG1) must exhaust the consequences of the 2-clauses before proceeding to resolve the 2-clauses

against the 3-clauses. D2, moreover, cannot prove W3 at all without some sort of level bound restriction, since otherwise it will get into a loop within the 2-clauses and generate

$$\bar{H}PPx \vee Fx,\ \bar{H}PPPx \vee Fx,\ \bar{H}PPPPx \vee Fx, \ldots .$$

Since the results described in this paper were obtained, several new strategies for limiting the number of resolvents generated have been proposed, most notably J. A. Robinson's 'P_1-deduction' (1965b) and B. Meltzer's 'P_p-deduction' (1966) and 'sharpened set of support' (Meltzer & Poggi 1966), which are all based on Robinson's 'P_1-deduction theorem', i.e., the theorem that the null clause can always be deduced from an unsatisfiable set C via a chain of resolutions R_1, R_2, etc., such that one parent of each R_i is 'positive' in the sense of containing no negated literals. Meltzer has shown that the predicates of an unsatisfiable set C can be renamed so that the set of positive clauses is in some sense minimal, and that this can usually be done so that all the positive clauses appear as a subset T' of T, where T is the set of clauses that formulate the special hypotheses of the proof and the negation of the conclusion. One may then use T' as a set of support, in conjunction with the restriction that one parent of each resolvent must be positive. This is a 'sharpened' set of support strategy, since T' is usually a proper subset of T, and since the restriction that one parent of each resolvent must be positive rules out some resolvents that the ordinary set of support strategy would permit. Preliminary investigation shows that T' can often be reduced to just one clause, as in our earlier example C1 (minus the superfluous first clause), wherein replacing P by $\bar{P}'$, $\bar{P}$ by P', D by $\bar{D}'$, and $\bar{D}$ by D' leaves (2) as the only positive clause. Examples given by Meltzer show that the 'sharpened' set of support strategy is capable of generating short proofs for many theorems, though few data have so far been obtained on the relative efficiency of this strategy in comparison with the ordinary set of support strategy with unit preference. The former strategy does rule out some inferences involving unit clauses that the latter strategy permits, e.g.,

$$\bar{P}x,\ Px \vee \bar{Q}x,\ \therefore \bar{Q}x$$

and the 'non-unit section' of one's program will therefore become relatively more important under the sharpened set of support strategy.

We have so far said nothing about the completeness of the methods discussed. D1 and D2 were not designed to be complete theorem-proving algorithms, but were intended rather as methods of producing efficient proofs for certain types of theorems. In order for them to be complete algorithms, it would be necessary to include an explicit 'non-unit section', as in PG1, general enough to provide for all cases in which the proof depends, in whole or in part, upon the generation of resolvents from pairs of non-unit clauses. The authors of PG1 have shown that their method is complete, so long as the level bound k is equal to or greater than the number of generations

$$S_0, S_1, S_2, \ldots, S_i, \ldots$$

that the original resolution algorithm would have to produce in order to obtain a proof, and so long as the set of support S is chosen in such a way that $C-S$ is satisfiable (i.e., such that the contradiction depends upon adding S to an otherwise consistent set of clauses). P_1-deduction and its variants are also provably complete. There is no necessary connection, however, between efficiency and completeness, and if and when really significant mathematical theorems are proven mechanically, the proofs will probably be generated by special-purpose heuristics rather than by theoretically complete general methods.

REFERENCES

Cook, S. A. (1965). Algebraic techniques and the mechanization of number theory, *RM*-4319-*PR*, RAND Corporation, Santa Monica, California.

Davis, M. (1963). Eliminating the irrelevant from mechanical proofs, *Proceedings of Symposia in Applied Mathematics* XV, pp. 15-30. American Mathematical Society, Providence, Rhode Island.

Gilmore, P. C. (1960). A proof method for quantification theory, *IBM J. Res. Devel.*, **4**.

Guard, J. R. (1964). Automated logic for semi-automated mathematics, *Contract No. AF*19(628)-3250, *Project No.* 5632, *Task No.* 563205, *Scientific Report No.* **1**, Applied Logic Corporation, Princeton, New Jersey.

Meltzer, B. (1966). Theorem-proving for computers: some results on resolution and renaming. *Comput. J.*, **8**, 341-343.

Meltzer, B., & Poggi, P. (1966). An improved complete strategy for theorem-proving by resolution, Internal report, Metamathematics Unit, University of Edinburgh.

Prawitz, D., Prawitz, H., & Voghera, N. (1960). A mechanical proof procedure and its realization in an electronic computer. *J. Assn Comput. Mach.*, **7**, 102-128.

Robinson, J. A. (1965a). A machine-oriented logic based on the resolution principle, *J. Assn Comput. Mach.*, **12**, 23-41.

Robinson, J. A. (1965b). Automatic deduction with hyper-resolution. *Int. J. Computer Math.*, **1**, 227-234.

Wang, H. (1964). Toward mechanical mathematics, *IBM J. Res. Dev.*, **4**. Also appears as Chapter IX in *A Survey of Mathematical Logic.* Peking: Science Press, and Amsterdam: North-Holland Publishing Company, 1964.

Wos, L., Carson, D., & Robinson, G. (1964). The unit preference strategy in theorem proving, *AFIPS Conference Proceedings*, **26**, 615-621. Washington, D.C.: Spartan Books.

Wos, L., Robinson, G. A., & Carson, D. F. (1965). Efficiency and completeness of the set of support strategy in theorem proving, *J. Assn Comput. Mach.*, **12**, 536-541.

MACHINE LEARNING AND HEURISTIC PROGRAMMING

6

AUTOMATIC DESCRIPTION AND RECOGNITION OF BOARD PATTERNS IN GO-MOKU

A. M. MURRAY and E. W. ELCOCK
UNIVERSITY OF ABERDEEN

INTRODUCTION

A series of computer programs have been written to play the board game Go-Moku. Go-Moku is played on a 19 × 19 square mesh. Player b(w) has a supply of black (white) pieces. The players take it in turns to play a piece on a mesh point. The winner is the first player to complete a *5-pattern*, that is, to make up a (horizontal, vertical or diagonal) line of five and only five adjacent pieces of his colour.

The programs carry out a backtrack analysis on games which have been lost to a human opponent. These analyses automatically generate a list of descriptions of subgoals, each hopefully describing the essential structure of a board pattern from which completion of a 5-pattern is inevitable. The playing capability of such a program depends critically on the power of the abstract descriptions and on the machinery for recognising realisations of described subgoals.

A distinct phase of the work was completed for the Machine Intelligence Workshop held under the auspices of the Experimental Programming Unit in Edinburgh in 1965. This phase included complete and working programs which allowed investigation and evaluation of a number of simple backtrack analysis routines. The learning programs resulting from this first exploratory phase automatically become competent players after a small number of games played against a human opponent. They reach a point, however, still far short of expert play, beyond which they cannot develop further. Their inadequacies did, however, highlight the problems that needed to be solved for more continuing self-improvement to be possible.

The paper by Elcock and Murray (1967) describing this work was published in the proceedings of the Workshop and provides useful background to the present paper.

The last year has been spent on exploring solutions of some of the problems highlighted by the earlier programs. A more sophisticated analysis routine which works in the context of a more powerful and general way of describing board patterns has been designed and implemented. The new programs are capable of analysing winning positions and, from the analysis, synthesising descriptions of subgoals of arbitrary complexity. An interesting method has been developed for automatically generating, from a description of a subgoal obtained by the analysis/synthesis process, a segment of control instructions which, when interpreted, direct the processing of the board required to recognise realisations of the described subgoal.

The present paper reports on the part of this work that is concerned with description and recognition.

DESCRIPTION OF BOARD PATTERNS

This section gives an informal discussion of the aspects of board patterns described and explains why the particular description scheme chosen is appropriate both to play of the game, and particularly to automatic acquisition of descriptions by program analysis of played games.

First, a few general remarks. A description says something about a more or less local pattern of played pieces on the board. It is *not* a representation: it does *not* attempt to say all that can be said about a particular local pattern in the sense that the particular pattern could be reconstructed from the description. The things a description does say express certain constraints (conditions) that a local pattern must satisfy in order to meet the description. The particular kinds of constraint that can be expressed are, hopefully, those that allow any particular board situation which contains a winning move to be described in a way which captures constructively the essential content, and only the essential content, which makes the win inevitable. Such a description, once acquired, should then be useable in game play to recognise any one of a class of board positions which in some local region meet the constraints of the description and which contain a recognisable winning move.

A description makes no reference to colour and can describe either a black object or a white object. In the text and diagrams that follow, white objects will be used throughout: corresponding black objects are obtained simply by replacing black by white and white by black.

Motivation of the descriptive scheme

The next few paragraphs try to expose, by means of examples, the motivation underlying the descriptive scheme implemented. In all diagrams only the relevant region of a board is given.

Example 1. In any of the board situations of Fig. 1 white play at node 1

completes a 5-pattern (and wins immediately). The essential content of a board situation which makes immediate completion of a 5-pattern possible is, trivially:

> there exists a node which is a constituent of a possible 5-pattern, of which 4 pieces have already been played, on some one line through the node

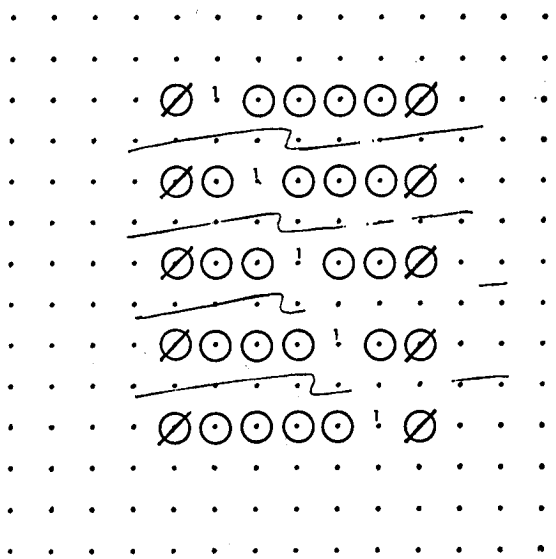


FIG. 1. Ø not a white piece.

Example 2. In any of the board positions of Fig. 2 white play at node 1 has the consequence that a 5-pattern can be completed at the next white move whatever the intervening black move: if black plays node 1′ (1″) then white plays 1″ (1′).

The essential content of all these board situations is:

> there exists a node which is a constituent of two possible 5-patterns, in each of which three pieces have already been played, on some one line through the node.

Attention is drawn to the fact that, as well as the node referred to explicitly, the object (as described by the text after 'there exists') contains, as it should, two implicit nodes (the nodes 1′ and 1″ of the representations of the object in Fig. 2).

It should be noted that, if it is the opponent to play, he can destroy a realisation of the object by play at any of its explicit or implicit nodes.

Consider the board situation:

Ø..:000:.Ø

The position contains two realisations of the object described above: one with α as the explicit node and one with β as the explicit node. These two

realisations have nodes in common (β is an implicit node for the realisation of which α is the explicit node . . .). With black to play he can destroy both realisations by playing at a common node.

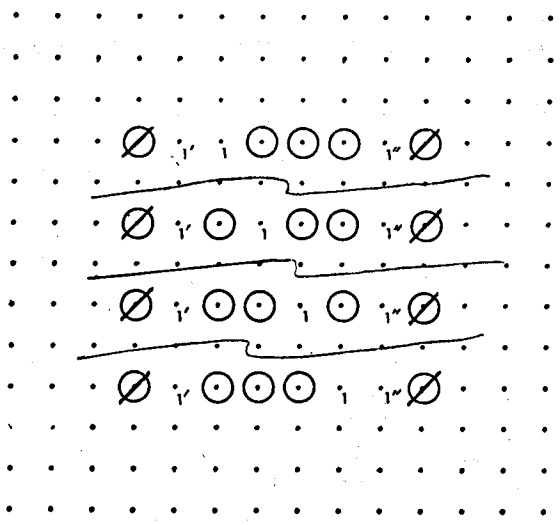


FIG. 2

Example 3. Fig. 3 shows a board position in which, after play at node 1, there are again two realisations of the object described in Example 2. This time, however, they do not share a common node; black cannot destroy both and white has a sequence of moves which inevitably lead to completion of a 5-pattern.

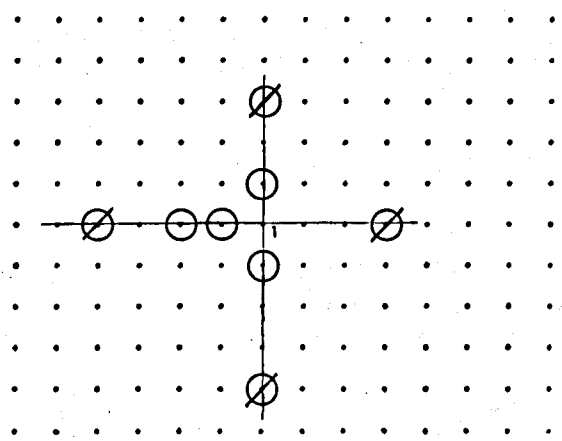


FIG. 3

The essential content of this (and similar) board positions is:

there exists a node which is a constituent of two possible 5-patterns, with two pieces played, on each of two lines through the node.

The description implies six other nodes; three on each line through the explicit node.

Play at the explicit node creates a set of realisations of the object described in Example 2 with no node common to all of them and which consequently cannot all be destroyed by the opponent's next move.

Example 4. Fig. 4 shows a board position in which, after play at node 1, there are two realisations of the object described in Example 1. A 5-pattern can always be completed in two player's moves.

The essential content of this class of board positions is:

> there exists a node which is a constituent of a possible 5-pattern, with three pieces played, on each of two lines through the node.

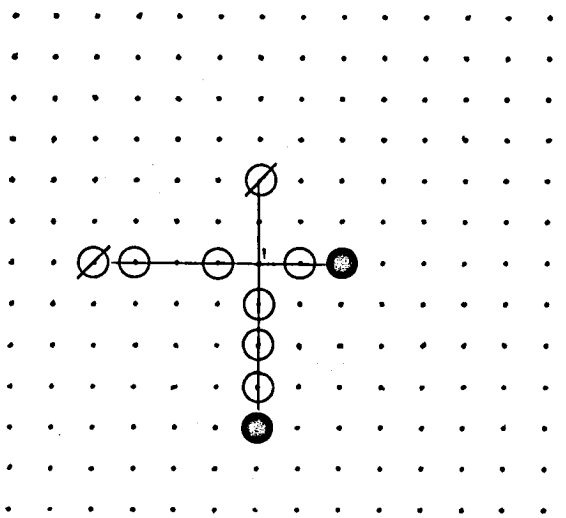


FIG. 4

Examples 5 and 6. So far the descriptions have been very simple: each has had just one explicit node and, in a realisation of the description, play at the explicit node initiates a sequence of moves leading inevitably to the creation of a 5-pattern.

Consider, however, the board positions of Figs. 5 and 6.

In the board position of Fig. 5, after play at node 1 a realisation of the objects described in Examples 2 and 3 is created at nodes 1′ and 2 respectively. Similarly, play at node 2 creates realisations of the same objects at nodes 1 and 2′ respectively.

The description (in words) of the essential content of this kind of board situation is rather lengthy:

> there exists a node (1) which is a constituent of two possible 5-patterns on each of two lines through the node: on one of the lines through the node the patterns have two pieces played: on the other line the patterns

have one piece played and a common implicit node (2) of these patterns is itself a constituent of two possible 5-patterns, with two pieces played, on some other line through it.

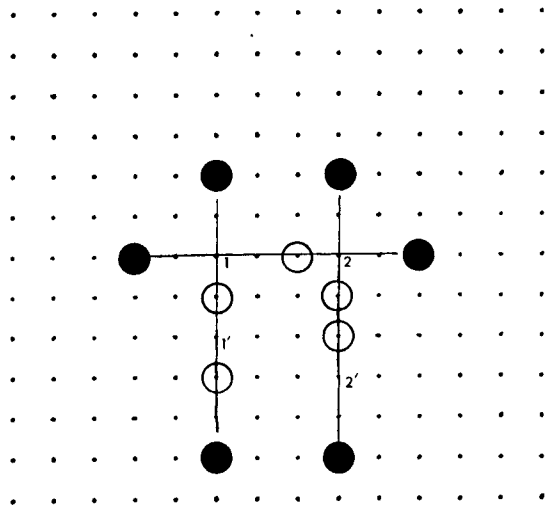


FIG. 5

The description makes explicit reference to two nodes. Nodes such as 1′ and 2′ of the realisation of diagram 5 are implied by the description.

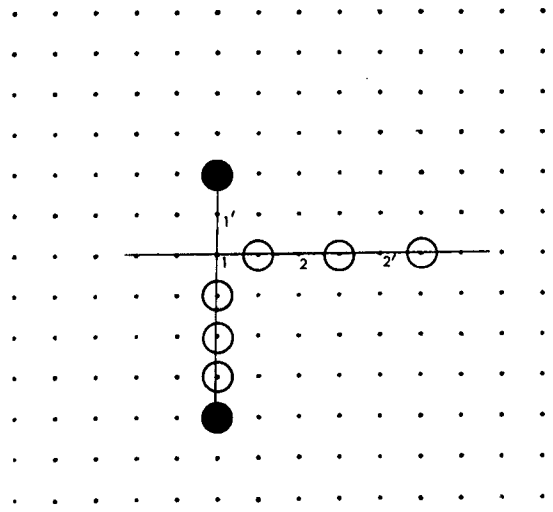


FIG. 6

In a realisation of this description the labelling of the explicit nodes 1 and 2 is arbitrary: the realisation has a certain symmetry. Play at either of the

explicit nodes of a realisation initiates a sequence of moves which leads to creation of a 5-pattern in at most four player's moves.

Contrast this with the board situation in Fig. 6.

After play at node 2, a realisation of the objects described in Examples 1 and 4 is created at nodes 2′ and 1 respectively.

The essential content of this kind of board situation is:

> there exists a node (1) which is a constituent of a possible 5-pattern on each of two lines through the node: on one of the lines the 5-pattern has three pieces played: on the other line the 5-pattern has two pieces played and an implicit node (2) of this pattern is itself a constituent of a possible 5-pattern on the same line, but with three pieces played.

The description again makes explicit reference to two nodes. This time, however, the labelling of the nodes 1 and 2 in a realisation is not arbitrary and, in fact, play at node 1 does not initiate a sequence of moves leading inevitably to the completion of a 5-pattern.

In the descriptive scheme implemented it is possible to add 'comments' to explicit nodes of a description. The 'comments' take the form of a statement of the expected number of moves required to complete a 5-pattern from play at a realisation of the object. As will be described later, these comments are made by program during the automatic acquisition of the description.

Automatic acquisition of descriptions

All the descriptions given in these examples imply that in a realisation of the description there exists a continuation which leads inevitably to the creation of a 5-pattern by the player. It is just the existence of this continuation that the description is designed to capture.

The described objects have this property because play at an appropriate node of a realisation creates a set of realisations of describable simpler objects, each of which has the property, and which do not share a common (implicit or explicit) node.

Descriptions with this property are called descriptions of 'subgoals'.

Most important is the implication of this property of the descriptive scheme for easy automatic acquisition of descriptions of subgoals by program.

Assume that the program can recognise realisations of described subgoals. Consider a stage in the program's game playing where the subgoals of Examples 2 and 3 have been described. Suppose the current state of the board in a game being played is that shown in Fig. 7. It is the opponent to play. At its last move the program did not recognise any realisations of described subgoals.

The opponent plays at node 1. The program now recognises the following realisations of known subgoal descriptions:

(i) a realisation of Example 2 at a with implicit nodes a' and b;
(ii) a realisation of Example 2 at b with implicit nodes a and b';
(iii) a realisation of Example 3 at c with implicit nodes $c', c'', c''', d', d'', d'''$.

An analysis of the situation might go as follows:

The realisations (i), (ii) and (iii) do not share a common node and therefore the opponent can inevitably create a 5-pattern.

A minimal set of realisations with this property is (i) and (iii). The piece at 1 must be a common played piece of each member of this set.

From this analysis it is reasonably straightforward to synthesise a description of the subgoal of which there was a realisation one move back in the game.

It is necessary to identify, for each subgoal realisation of the minimal set, the relevant lines for which the piece at 1 is a played piece. It is then necessary to (i) downgrade (by effectively unplaying this piece) the components of the description that refer to these lines, and (ii) make node 1 an explicit node of

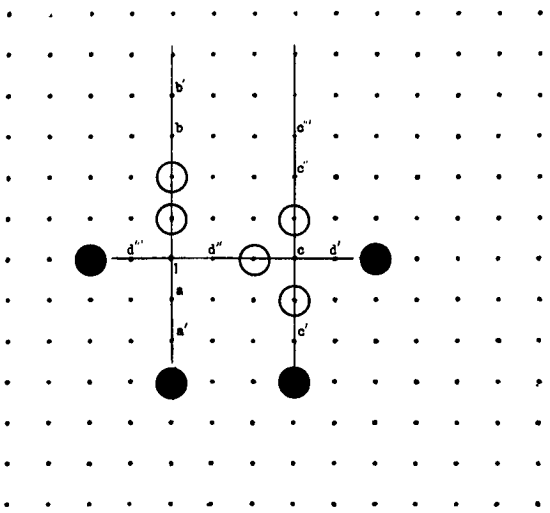


FIG. 7

these downgraded descriptions. A new description can now be synthesised from these downgraded descriptions by taking the union of them through the common node 1.

This description will describe a subgoal for which there is the realisation in the particular board situation from which the description was abstracted. Once the description is acquired, any of the whole set of its realisations can be recognised in future games.

It is not difficult to make this synthesis procedure precise and to automate it so that the program can, from game play such as the above, acquire descriptions of new subgoals. By the very nature of the process, this acquisition of descriptions of subgoals gives the program an increasing capability for directed play in board situations further and further away from a win.

RECOGNITION OF BOARD PATTERNS

The end result of being able to acquire descriptions of subgoals is to be able to recognise realisations of them in game play. This section discusses briefly

how this is done. Since one of the main interests here is the program implementation of the recognition process, it is necessary to introduce a notation for descriptions which is closer to the program level than the segments of English text given in the examples in 'motivation of the descriptive scheme'.

A formal notation for descriptions

As we have seen, a description specifies a number of explicit nodes and certain linear 5-pattern relationships between them. The nodes are labelled 1, 2, 3 ... A description consists of a 'node list' followed by a 'pattern list'.

The node list has an element for each explicit node (the 'parent' node of the element). The elements are listed in the serial order of the labels of their parent nodes. Thus

$$(1, 2, 3, 4-1, 3/4-1, 2/4-1, 2/3)$$

is the node list for a subgoal with 4 nodes, the elements of the node list being separated by '—'.

Each element on the node list is a 'line list' with up to four elements each referring to a different line through the parent node, elements of the line list being separated by ','. An element consists of the labels of those and only those explicit nodes which, in common with the parent node, contribute to a possible 5-pattern or patterns, specified in the pattern list, on the line. The label of the parent node itself is suppressed unless there would otherwise be a null entry.

Thus, the first element of the node list above states that node 1 is a node of specified (in the pattern list) patterns on four lines through it. On one line no other nodes are referenced. Nodes 2, 3 and 4 are referenced on one each of the other three specified lines. The second element states that node 2 is a node of specified patterns on two lines through it. On one line node 1 is referenced and on the other line nodes 3 and 4 are referenced. The third element, etc.

It should be emphasised that the node list specifies the line pattern intersections completely, i.e., none of the specified line patterns intersect in a node of any other specified pattern other than at the stated explicit nodes.

The pattern list has the same structure as the node list. There is an entry for each line element of the node list. The entry specifies the possible 5-pattern or patterns, of which the parent node is a constituent, that must exist on the line. The patterns are specified by an integer $n (0 \leqslant n \leqslant 5)$ equal to the number of pieces in the pattern after play at the parent node. If the node must be a constituent of two patterns with the given n, then this is indicated by dashing the value of n (e.g. 4').

Consider the complete description:

$$(1, 2, 3, 4-1, 3/4-1, 2/4-1, 2/3)$$
$$3', 2', 2', 2'-2', 1'-2', 1'-2', 1'$$

Node 1 must be a constituent of two possible 5-patterns, in each of which two pieces have been played, on a line through it which references no other nodes. It must be a constituent of a 2' on each of the three other lines through it

passing through nodes 2, 3 and 4 respectively. Node 2 must be a constituent of a 2′ on the line through it which passes through node 1, and a constituent of a 1′ on another line through it which passes through nodes 3 and 4, etc. A realisation of the described object is given in Fig. 8.

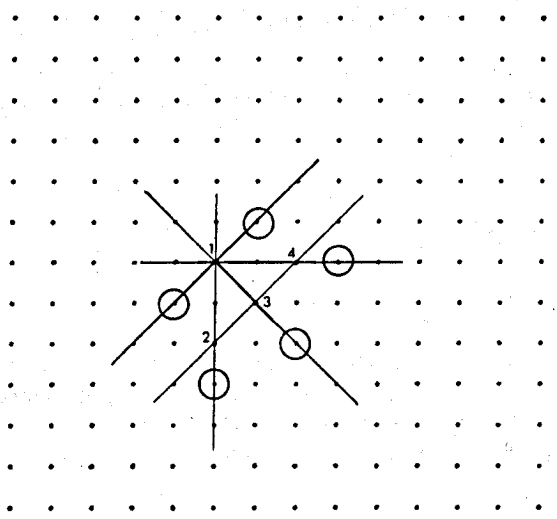


FIG. 8

Finally, comments may be added. As already mentioned (p. 81), the only comment for which provision is made in the present implementation is the prefixing of an element of the pattern list by a number equal to the expected number of player's moves needed to complete a 5-pattern in a realisation of the subgoal. Thus, the comment '6:' in

(1, 2, 3, 4−1, 3/4−1, 2/4−1, 2/3)
6: 3′, 2′, 2′, 2′−2′, 1′−2′, 1′−2′, 1′

for the subgoal of diagram 8.

A complex subgoal can be described in different equivalent ways corresponding to different labelling of nodes, etc. An arbitrary set of rules is used to ensure that descriptions of essentially the same object will always be given in a unique 'canonical' form.

In canonical form, the descriptions of the subgoals of the examples in 'Motivation of the descriptive scheme' are (note that subgoals having the same node list are grouped together):

(1)	
1:5	diagram 1
2:4′	diagram 2
(1, 1)	
2:4, 4	diagram 4

3:3′, 3′	diagram 3
.	
(1, 2−1)	
4, 3−3:4	diagram 6
.	
(1, 2−1, 2)	
4:3′, 2′−4:2′, 3′	diagram 5

The recognition processor

What has been given above is essentially an external (written) notation for descriptions. The program uses an equivalent representation in terms of a list structure. The written notation, or its equivalent list representation, exposes the structure of the subgoal and is particularly appropriate for the manipulations involved in the synthesis of new descriptions. It does not, however, make clear in any operational sense the sequence of actions that must be gone through to recognise a realisation of the described object in any particular board situation.

Recognition is facilitated by the use of a quite different representation of the description called the 'control stream' representation. Each control stream is made up of a sequence of control elements. Each control element in effect specifies a board processing instruction. The complete sequence of elements specifies the total processing that has to be gone through to recognise a realisation of the described object or, more precisely, to determine whether or not a local region of the board contains a realisation of the described object with a particular board node identified with node 1 of the object.

In what follows an example of a control stream is discussed informally and without going into too much detail. The intention is simply to bring out the difference between the synthetic and operational descriptions of subgoals and to present the view that the data structure which is the control stream can be thought of as a set of instructions in a board processing language.

Fig. 9 shows again a realisation of the subgoal of Fig. 8 and gives the written description together with the equivalent control stream. The format of the control stream is a feature of a particular machine implementation and is irrelevant to the present discussion.

In processing a particular board situation for realisations of the subgoal a particular vacant mesh point on the board is assigned the label "1" (i.e., is associated with node 1 of the description), and the control scheme is scanned by an interpreter.

An 'A' element has the structure:

(A code; pattern list element).

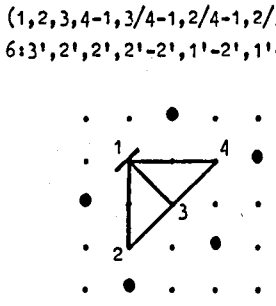
The first control stream element of the example of Fig. 9 is interpreted as 'check that mesh point "1" is a constituent of possible patterns meeting the specifications 3′, 2′, 2′ and 2′ on different lines through the mesh point: if successful, continue the scan'.

A 'B' element has the structure:

(B code; line pattern specification; node label; line label).

The second element of the example is interpreted as 'look for a line through the mesh point "1" for which the mesh point is a constituent of a pattern which meets the specification 3′; if successful, call this line "line 1 of node 1" and continue the scan'.

The third element of the example is interpreted as 'look for a line through the mesh point "1", not currently called "line 1 of node 1", for which the mesh point "1" is a constituent of a 2′: if successful, call this line "line 2 of node 1" and continue the scan'.



A	3'	2'	2'	2'				
B					3'			1,1
B					2'			1,2
C	2'	1'			2'	2		1,2
B					2'			1,3
C	2'	1'			2'	3		1,3
D				1'	1'	3,2		2,2
B					2'			1,4
C	2'	1'			2'	4		1,4
D				1'		4,2		2,2
D						4,1		3,2
E					3'			1,1
E	2				3'			1,2
E	3				3'			1,3
E	4				3'			1,4
E	4	3			3'			2,2

FIG. 9

A 'C' element has the structure:

(C code; pattern list element; line pattern specification; node label; node label; line label).

The fourth element of the example is interpreted as 'look for a vacant mesh point on line 2 of node 1, lying within a possible 5-pattern of which mesh point 1 is a constituent, which is a constituent of possible patterns meeting the specifications 2′ and 1′ on different lines through the new mesh point and which in particular meets the specification 2′ along line 2 of node 1: if successful, assign the label "2" to the mesh point and continue the scan'.

D-words are concerned with checking that already allocated nodes which are required to lie on a line are in fact on a line, and that the pattern specifications in the direction of this line are met.

E-words are concerned with finally checking that, for each line, the allocated nodes are in fact nodes of the same possible 5-pattern (the C-word allocations do not ensure this); checking the specification of this pattern, and listing implicit nodes (defensive moves). If the action of the last E-word is successful then a realisation of the subgoal has been recognised.

If at any stage it is impossible to continue the forward scan either because it is impossible to make the required allocation of a node or line, or after the last E-word to find other realisations with the same mesh point 1, the control

stream is backscanned by the interpreter to the first significant new allocation and the forward scan restarted.

CONTEXT AND SUMMARY

So far nothing has been said about how the control stream representation of subgoals is generated. In fact, there is a component of the total program which can transform the 'written' description, generated by the synthesis procedure, into the equivalent control stream.

With this in mind, Fig. 10 gives a schematic representation of the total program that has been written and of which some aspects of the pattern

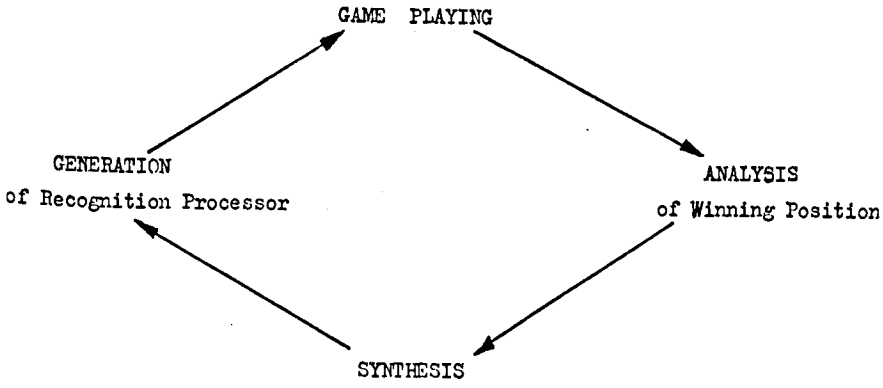


FIG. 10

description and recognition components have been discussed here. We would like to stress the view that the program does not generate data in the conventional sense, but rather segments of subgoal recognition program.

Finally, to counterbalance the simplicity of the examples treated on pages 83-86, Fig. 11 gives subgoal descriptions (together with graphic mnemonics) which present a typical list generated by program over a sequence of played games. It should perhaps be added that, with this list, the program plays at expert level.

REFERENCE

Elcock, E. W., & Murray, A. M. (1967). Experiments with a learning component in a Go-Moku playing program. *Machine Intelligence* 1, N. L. Collins and D. Michie (eds.), pp. 87-103. Edinburgh: Oliver and Boyd.

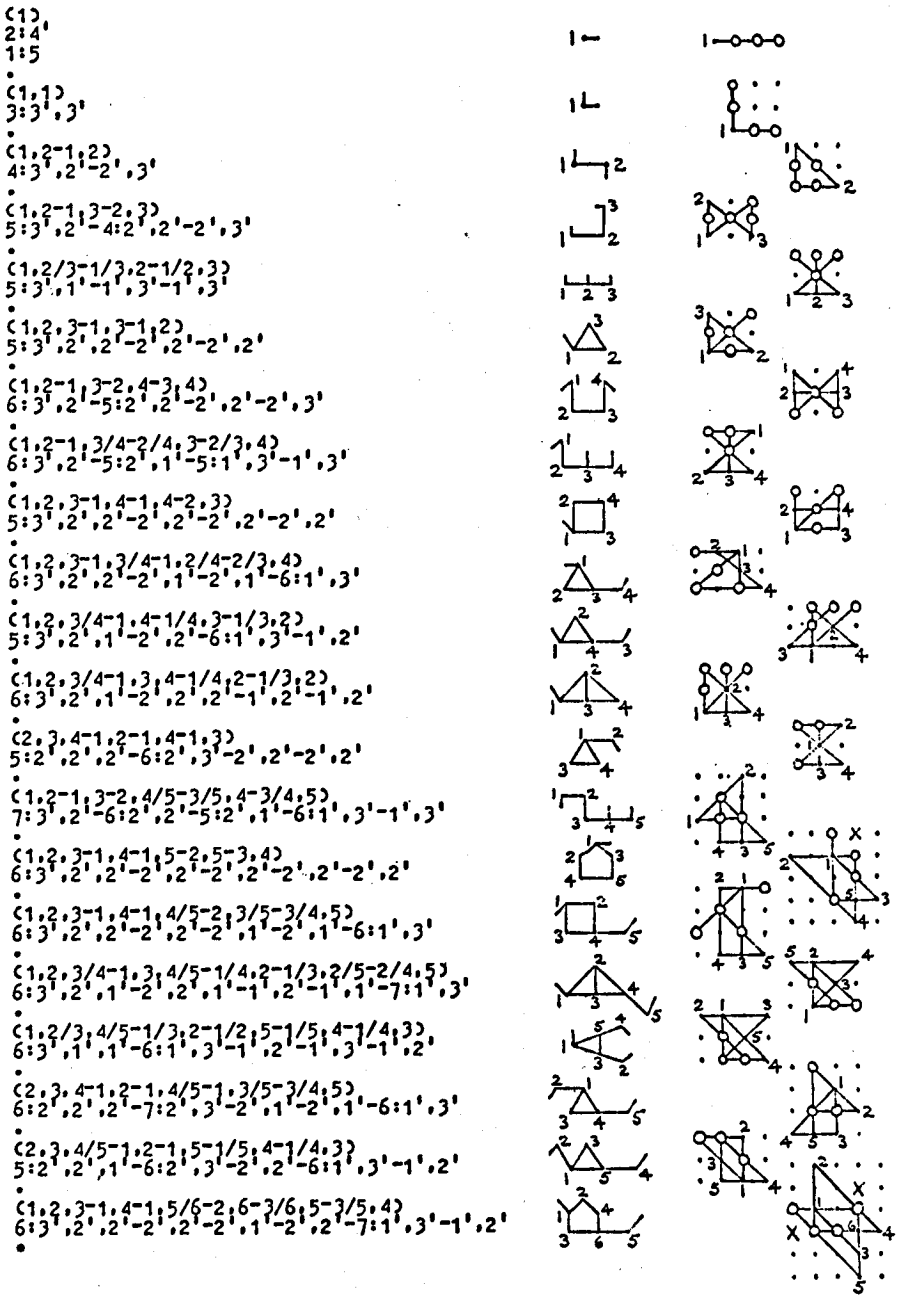

(1)
2:4'
1:5
(1,1)
3:3',3'
(1,2-1,2)
4:3',2'-2',3'
(1,2-1,3-2,3)
5:3',2'-4:2',2'-2',3'
(1,2/3-1/3,2-1/2,3)
5:3',1'-1',3'-1',3'
(1,2,3-1,3-1,2)
5:3',2',2'-2',2'-2',2'
(1,2-1,3-2,4-3,4)
6:3',2'-5:2',2'-2',2'-2',3'
(1,2-1,3/4-2/4,3-2/3,4)
6:3',2'-5:2',1'-5:1',3'-1',3'
(1,2,3-1,4-1,4-2,3)
5:3',2',2'-2',2'-2',2'-2',2'
(1,2,3-1,3/4-1,2/4-2/3,4)
6:3',2',2'-2',1'-2',1'-6:1',3'
(1,2,3/4-1,4-1/4,3-1/3,2)
5:3',2',1'-2',2'-6:1',3'-1',2'
(1,2,3/4-1,3,4-1/4,2-1/3,2)
6:3',2',1'-2',2',2'-1',2'-1',2'
(2,3,4-1,2-1,4-1,3)
5:2',2',2'-6:2',3'-2',2'-2',2'
(1,2-1,3-2,4/5-3/5,4-3/4,5)
7:3',2'-6:2',2'-5:2',1'-6:1',3'-1',3'
(1,2,3-1,4-1,5-2,5-3,4)
6:3',2',2'-2',2'-2',2'-2',2'-2',2'
(1,2,3-1,4-1,4/5-2,3/5-3/4,5)
6:3',2',2'-2',2'-2',1'-2',1'-6:1',3'
(1,2,3/4-1,3,4/5-1/4,2-1/3,2/5-2/4,5)
6:3',2',1'-2',2',1'-1',2'-1',1'-7:1',3'
(1,2/3,4/5-1/3,2-1/2,5-1/5,4-1/4,3)
6:3',1',1'-6:1',3'-1',2'-1',3'-1',2'
(2,3,4-1,2-1,4/5-1,3/5-3/4,5)
6:2',2',2'-7:2',3'-2',1'-2',1'-6:1',3'
(2,3,4/5-1,2-1,5-1/5,4-1/4,3)
5:2',2',1'-6:2',3'-2',2'-6:1',3'-1',2'
(1,2,3-1,4-1,5/6-2,6-3/6,5-3/5,4)
6:3',2',2'-2',2'-2',1'-2',2'-7:1',3'-1',2'

FIG. 11

7

A FIVE-YEAR PLAN FOR AUTOMATIC CHESS

I. J. GOOD*
TRINITY COLLEGE, OXFORD, and
ATLAS COMPUTER LABORATORY, CHILTON

JUSTIFICATION OF CHESS PROGRAMS

Young animals play games in order to prepare themselves for the business of serious living, without getting hurt in the training period. Game-playing on computers serves a similar function. It can teach us something about the structure of thought processes and the theory of struggle and has the advantage over economic modelling that the rules and objectives are clear-cut. If the machine wins tournaments it must be a good player.

The complexity and originality of a master chess player is perhaps greater than that of a professional economist. The chess player continually pits his wits against other players and the precision of the rules makes feasible a depth of thinking comparable to that in mathematics.

No program has yet been written that plays chess of even good amateur standard. A really good chess program would be a breakthrough in work on machine intelligence, and would be a great encouragement to workers in other parts of this field and to those who sponsor such work.

In criticism of the writing of a chess program, Macdonald (1950) quoted a remark to the effect that a machine for smoking tobacco could be built, but would serve no useful purpose. The irony is that smoking machines have since been built in order to help research on the medical effects of smoking. This does not prove that a chess program should be written, but suggests that the arguments against it might be shallow. Many branches of science, and of pure and applied mathematics, have started with a study of apparently frivolous things such as puzzles and games.

* Now at Virginia Polytechnic Institute, Blackburg.

It is pertinent to ask in what way a good chess program would take us beyond the draughts program of A. L. Samuel (see Appendix B). The answer is related to the much greater complication of chess, the much larger number of variations and possible positions. In fact, the number of possible chess positions is about the cube or fourth power of the number of possible draughts positions (see Appendix E). Samuel was able to make considerable use of the storage of thousands of positions that had occurred in the previous experience of the machine, and this led to a very useful increase in the depth of analysis of individual positions. The value of this device depends on the probability that, at any moment in the analysis, we run into a position that has already been analysed and stored. If the expected number of previously analysed and stored positions on an analysis tree exceeds unity, then the expected *effective* number of positions on the tree is infinite! This remark is of course based on an over-simplified model, as is clear since the total number of possible chess or draughts positions is finite. The remark is also somewhat misleading, since an 'infinite' tree might be deep at the wrong places; but it does give a little insight into the effect of storing positions. (See also (*a*) on p. 92 and Appendix H, p. 114.) Whereas it is useful to store thousands of positions in draughts, in chess it would be necessary to store millions or billions of positions in order to gain a comparable advantage. Presumably the main advantage of storing positions in draughts accrues from the opening and early middle-game; and in chess, by storing millions of positions, the machine could gain an opening advantage. Apart from this, I have believed for a very long time that a good chess program would need to make use of essentially the same methods as those used by men. The difficulty, the interest, and the challenge, is to formalise exactly what it is that men do when they play chess.

The more the program is based on the methods used by humans the more light it will shed on the nature of thought processes. But for the sake of a clear-cut objective I should like to write a program that wins games.

When a man plays chess, he does often recognise situations that have occurred before, but these situations are seldom complete positions, except in the opening and very late end-game; rather they are features of the position, or patterns embedded in it. Thus chess provides an example of pattern recognition, whereas draughts does not to much extent. Nevertheless, several of the techniques used by Samuel should be incorporated.

Chess also provides a better example than draughts of the use of associative memory, since any given position is associated with many situations that have occurred in the chess player's experience. These associated situations suggest strategic or tactical ideas to the player, the strength of the suggestions being dependent on the strength of the associations.

An example of a pattern is when White's king has three unmoved pawns in front of it in the late middle-game or early end-game. This pattern is recognised by every experienced chess player as potentially dangerous, in that there is the possibility of a mate by a rook or queen on the back line,

even in conjunction with a sacrifice. *One* analysis of the position will then be performed with this theme in mind. This applies more generally to the goals and subgoals that occur to the chess player. Thus there should be 'specifically-directed' as well as 'routinely-directed' analysis.

Another important aspect of chess thinking, also required in most other problem-solving, is what de Groot (1946, 1965) calls 'progressive deepening' of an analysis. Typically an analysis of a position by a human player does not simply follow a tree formation, but contains cycles in which a piece of analysis is retraced and improved. (Compare the not purely hierarchical structure of definitions and perception discussed by Good 1962a, p. 124, and 1965a, pp. 42 and 75.) Also a player continually formulates subgoals and subsubgoals and continually modifies them. In these respects again chess provides a better example than draughts.

As every philosopher and programmer knows, it is not easy to formalise human thought. In fact, when we say that a man has made a judgement, we imply that we do not know in detail how he did it, and this is so even if that man is ourselves (Good 1959). This aspect of judgements is not yet mentioned in dictionaries. The work on automatic chess therefore started when men, such as the musician-chess player Philidor (1749), formulated general principles of play. Many of these principles have become embodied in chess slogans or clichés that need to be taken into account by the programmer (see Section 4.2).

Many of the principles of chess are expressed linguistically rather than numerically. This is much more typical of chess, and of many intellectual activities in ordinary life, than in draughts. Thus chess provides a good field, with a limited vocabulary, for those who are working on language handling by computers. (An example that did occur in draughts, or should have occurred, is mentioned in Appendix B, and curiously enough was overlooked by the strong draughts player who was pitted against a machine. He could have drawn the game, and the method can be readily expressed in words, but only artificially in numerical terms.)

Chess programming can also be justified as sport. £500,000,000 are spent on football pools in the United Kingdom in five years, and each pound must correspond to several hours of study. Further, hundreds of millions of man-hours are expended in watching the game. The expenditure on chess programming will be microscopic in comparison.

The nature of complex games

There is one principle of chess that is so old that the early theoreticians probably did not bother to mention it. It is certainly as old as the game of Go, at least 3000 years, and is typical of all complex games, although stochastic games have additional features. It is the habit of thinking 'if I go there, then he might go there or there, and, in the first case, I might go there, etc.'. In other words, analysis often follows the branches of a tree, with evaluations at the endpoints, followed by back-tracking or iterative minimaxing (see

Appendix A). This process corresponds closely to the formulation of games in 'extensive form' (see von Neumann & Morgenstern 1947, or Luce & Raiffa 1957, Chapter 3). The notion of a game tree applies of course much more generally than to chess. With regard to the definition of the endpoints, it is sometimes said that they should be quiescent positions, but, provided that the outcome of the game is clear enough, an endpoint can also rationally be a turbulent position. For this reason, it is convenient to introduce a notion called 'agitation', which is a modification of 'turbulence' taking into account the probability that the result is clear-cut. (See Appendix H for a discussion of the terminology and its relationship to decision making in general.)

Let us suppose that we can measure, for any position π, both its *turbulence* and also the *superficial probabilities* that the player can win, draw, or lose, p_W, p_D, and p_L. (By a 'superficial probability' I mean a probability based on some evaluation function, without any forward analysis. Perhaps 'surface probability' would be a better term.) Then the decision of whether to regard that position, π, as an endpoint, when analysing a position, π_0, depends on the following considerations:

(*a*) The *depth* (i.e., the number of moves ahead) of the position π from the current position, π_0. More precisely, the decision depends on the probability that we shall reach π from π_0. The smaller this probability, $P(\pi|\pi_0)$, or, less accurately, the greater the depth, the more rational it is to treat π as an endpoint of the analysis tree for π_0. The 'probabilistic depth' of π from π_0 could reasonably be defined as proportional to $-\log P(\pi|\pi_0)$, and the expected amount of information in an analysis, or its effective depth could be defined as $-\sum_\pi P(\pi|\pi_0) \log P(\pi|\pi_0)$ summed over all the endpoints of the tree. This is an incomplete entropy, since, usually, $\sum_\pi P(\pi|\pi_0) < 1$. This definition suggests itself, but I think the analysis given in Appendix H will turn out to be more useful. The value of storing previously analysed positions could perhaps be estimated in terms of 'effective depth'.

(*b*) The lower the turbulence of π, the more prepared we should be to treat it as an endpoint, other things being equal.

(*c*) The more obviously the outcome of the game is decided at π, the more prepared we should be to treat π as an endpoint, other things being equal.

(*d*) The larger the analysis tree as a whole, as far as we can judge, the more prepared we should be to treat π as an endpoint, other things being equal.

(*e*) The less time we have left on our clock, the more we should be prepared to treat π as an endpoint, other things being equal.

(*f*) The less time our opponent has left on his clock, the more prepared we should be to treat π as an endpoint, other things being equal. (This advice is double-edged!)

In other words, we should define a certain function of six variables, monotonic in each separately, and should treat π as an endpoint if this function exceeds some threshold.

Notice the following philosophical point. The superficial probabilities are not strict logical probabilities, since they are based on a suppression of a

logical argument, namely a forward analysis. (Compare the discussion of the probability of mathematical propositions in Good 1950a, p. 49.) Since we are forced to make use of superficial probabilities, there is bound to be an element of luck in chess. (There has even been a competition on luck in chess: see Golombek 1966.) From the point of view of strict logic, chess is a game of pure skill, but it is a physical impossibility to avoid some degree of chance. Just as in mathematics we must make use of probabilities based on incomplete analysis. When trying to prove a mathematical theorem, it is necessary to formulate subgoals, and to estimate superficial probabilities for them, in order to find an appropriate strategy. Theorem-proving resembles chess playing in that we have an objective and an analysis tree, or graph, but differs in that a superficial expected pay-off replaces the iterated minimax. The minimax idea can come in if we are trying to prove a theorem and we imagine that we have an opponent who wishes to disprove it. The value of our game is 1 if the theorem is true and -1 if it is false. In the proof trees described in the paper by Dr D. C. Cooper the 'and's correspond to moves of the opponent, since we must allow for both branches, whereas the 'or's correspond to our own moves. The minimax (strictly maximin) value of the tree tells us whether the theorem is true, and, if we allow for superficial probabilities at the end-points of the tree, the minimax value is the superficial probability of the theorem.

The programming of complex games will also exemplify an aspect of practical decisions, since here too it is typical that we are forced to make use of superficial probabilities.

Closely allied to the problem of selecting endpoints on an analysis tree, is the selection of which moves to discard without analysis. This problem is discussed in Appendix H.

SOME POSSIBLE KINDS OF CHESS PROGRAMS

2.1. Chess is a game of perfect information, and is a finite game if the 50-move drawing rule is mandatory (see Appendix D). Hence, from the point of view of the Borel-von Neumann theory of games, chess is 'trivial' (see Appendix A). This shows that, for practical chess, the Borel-von Neumann theory has trivial relevance. But it would be a good exercise to program a computer to do an exhaustive analysis of any position given to it. Such a program could be used for solving two-movers, a program for which has already been written at M.I.T., but it would have little applicability to normal chess. It could be largely written independently of the rules of chess, and could invoke these rules as a sub-routine. It would then have applications to other games and decision problems. If this program is written it should have the option of being stopped at any assigned depth.

A program for solving ordinary 'two-movers' would be useful when dealing with a large number of problems, for a tournament or book, or for a writer of chess columns in newspapers. Professor R. C. O. Matthews points out (private communication) that it could be used in the composition of problems.

2.2. If our object is to learn as much as possible about learning, we could try to program a machine to learn the game without even telling it the rules. This is how Capablanca learned chess: by watching his father and uncle playing. Incidentally he won the first game he played, at the age of 6, against his uncle. McCulloch (1965, p. 199) calls a machine of this kind an 'ethical' machine. It would need to be able to formulate a very wide class of hypotheses.

2.3. Next, we could give the rules to the machine and let it learn the principles for itself. Here again the machine would need to be able to formulate a wide class of hypotheses, but not as wide as the 'ethical' machine.

2.4. Next, we could give the machine a limited class of principles and allow it to work out its own evaluation functions in terms of these principles, in the light of its experience. The experience could either be the playing of games or, more efficiently, we could tell the machine which moves in various positions were good or bad, and how good or bad. The simplest form of evaluation function would be linear, with coefficients that were allowed to be positive, negative or zero. The optimisation of these coefficients constitutes a primitive form of learning, but I think most people would be reluctant to call it 'concept formation'. If however non-linear polynomials are permitted, or equivalently if logical combinations of the principles can be formed and incorporated into the linear function, then concept formation becomes possible. For example, a machine might have discovered that two bishops are usually better than bishop and knight, it if had been programmed to examine quadratic evaluation functions (Good 1959). After this had been established, it might have gone on to modify the concept by saying that it was true in open positions, but not in closed positions. It might have discovered the whole of this principle in one step if it had been allowed to use cubic evaluation functions, but this would be a more expensive way of making the discovery. It seems reasonable to say that *the use of quadratic evaluation functions provides the first step in automatic creativity or concept formation.*

Humans usually think this way. First they give relative weights to features, then they look for interactions between pairs of these features. When they find a pair of features that need to be taken together, they define this as a new feature. Thus high-order interactions are often discovered piecemeal, but of course this is not always possible.

The discovery of more intricate concepts will require the generation of a large number of propositions, and the order in which they should be generated provides a most interesting search problem.

2.5. We could try to write the program with man-machine synergy in mind from the start (regarding the terminology, see Appendix K). We would aim to teach the machine how to play chess, and also to learn something about chess from the machine. Our long-term aim would be to cooperate with the machine in games against grand-masters, against other machines, and against other synergistic combines. A simple and effective example of man-machine

synergy is provided by R. C. O. Matthews' suggestion mentioned in section 2.1.

2.6. We could go all out to make the machine a good chess player, by putting as much as we can into the low-level program, and without relying on more machine learning than is implied in the optimisation of linear forms.

2.7. Same as 2.6, but with the machine capable of forming new concepts, that is, allowed at least to use non-linear evaluation functions.

Of these various aims, which are not mutually exclusive, the one under heading 2.2 is the most exciting, but 2.6 would be easier. I think a reasonable approach is to aim at 2.6 at first, in the hope of getting a good chess program, but always holding the other aims in mind.

A BRIEF HISTORY OF CHESS PROGRAMS

As mentioned earlier, the formalising of principles for playing chess began with the early theoreticians. But they wrote exclusively for people, just as grammarians wrote for people and not for machines until the advent of computational linguistics. Emanuel Lasker (1932, p. 339) said 'It is easy to mould the theory of Steinitz into mathematical symbols by inventing a kind of chess, the rules and regulations of which are themselves expressed by mathematical symbols. The Japanese game of Go is very nearly what I mean. In such a game the question, whether thorough analysis would confirm the theory of Steinitz or not, presumably could be quickly solved because the power of modern mathematics is exceedingly great.' What Lasker had in mind was clearly not a mathematical method of playing chess, but a mathematical demonstration of the validity of strategic principles for wider applications. Incidentally, I believe that a good Go-playing program would be much more difficult to write than a good chess program, because Go, not to be confused with Go Moku, depends so much on judgement. Perhaps Lasker was misled by the charming simplicity of the rules of Go (see Good 1965b).

In about 1932, one of my schoolboy chess associates named G. T. Hammond pointed out to me, without claiming originality, that there is a simple mechanical method of playing that is not as bad as one would expect, at any rate in the opening. The method is to play the move that maximises one's *mobility*, where mobility is defined as the number of possible legal moves. I cannot remember whether he allowed for the mobility of the opponent. We realised that this rule should not be used if it involved putting a piece *en prise*, and that the rule was not much good in a combinative position. But this observation about mobility is I think the most surprising single fact, so far, about automatic chess. It might date back much further than 1932, since the theoretical valuation of the pieces, of 1876, was based on the average numbers of squares controlled (see Appendix F). Compare also a remark attributed to Emanuel Lasker by Edward Lasker (1951, p. 56), '. . . the player who gains control of more than 32 squares has the better winning chance'.

Good (1939) remarked that it might be possible to translate into a mathe-

matical technique the knack that masters have of picking out only those lines that matter from the millions of possible ones. This is of course the central problem in writing a good chess program. (When that note was written, what is now called 'programming' would have been called 'mathematics'.) Another point in that little article was that some statistical reasoning could shed light on the nature of chess. For example, White's advantage in moves is $1-1+1-\ldots$, which is summable $(C, 1)$ to $\frac{1}{2}$. In other words, White is, on the average, half a move ahead. Since $2\frac{1}{2}$ moves are worth a pawn, it follows that White's advantage is about a fifth of a pawn. Now, by statistical sampling we could quantify what I called the law of multiplication of advantage, and, from this, we might be able to shed light on the question of whether White has a theoretically forced win.

In 1940, when I met and worked with Turing, we had several discussions on how chess playing could be formalised. We also later had discussions with Shaun Wylie and Donald Michie. Essentially similar ideas were published by Shannon (1950), who had been thinking independently about chess programming. Turing, Michie and Wylie, in 1947-48, designed some one-move analysers, a discussion of which was published by Maynard Smith and Michie (1961) and Michie (1966).

In about 1941, I had discussions with J. Gillis on the possibility of founding a theory of imperfect logic, in which propositions would be exponentially forgotten if not used for some time, and also could be reinforced (the aim was to formalise human logic). A similar idea has been used very effectively by Samuel in his draughts-playing program.

Perhaps the most intensive work ever done on the formalising of chess thinking was that of de Groot. He studied the psychology of chess playing for over twenty years (see de Groot 1946, 1965). I have not yet been able to judge how much of the later literature is explicit in de Groot's writings. His basic experimental method was to record the spoken moment-to-moment thoughts of chess masters.

Shannon (1950) made the following points:

(*a*) Different routines should be used for opening, middle-game and ending.

(*b*) Forceful variations should be analysed out as far as possible and evaluated only at points of some stability.

(*c*) Select the variations to be explored by some process so that the machine does not waste its time on totally pointless variations. (This point was covered by Good 1939.)

(*d*) The threshold for deciding whether a position should be regarded as stable should depend on the number of moves advanced from the current position.

(*e*) A statistical element can be introduced in the selection of moves so that an opponent who has won once cannot win again by exactly the same sequence of moves.

(*f*) A few hundred openings could be stored and played by rote.

(*g*) A learning program could be based on the use of a higher-level program

that changes the terms and coefficients in the evaluation function, according to the results of games.

Slater (1950) argued that 'it does seem possible that a chess computer which was programmed, beyond immediate tactical tasks, to maximise the mobility difference between itself and its opponent over a series of moves, might play a strategically tolerable game of chess'. His thesis was based in part on an examination of master games. (I do not know whether G. T. Hammond's remark was so based.) He also gave some statistics relevant to the law of multiplication of advantage.

Good (1950b) pointed out that 'the analysis of forceful variations in any position would be like a stochastic branching process, and the endpoints of the corresponding tree would be quiescent positions needing strategic evaluation (but see Appendix H). The total time taken to cope with such a tree would be roughly proportional to the number of individuals in this tree, and it is known that the probability distribution of this number is extremely skew (see Good 1949). . . . There would therefore be a danger of the machine running into time trouble. It would then modify its definition of the forcefulness threshold of a move.'

Good (1959, based on a lecture of January 1958) pointed out that:

(*a*) The training of a machine should be move by move, and not merely by the results of games.

(*b*) In quiescent positions it should be possible to find some rule for estimating the probability of a win, draw or loss, and this is analogous to the estimation of the probability of a mathematical theorem. 'It may be sufficient to assume that the log-odds are linear in all the types of advantage that are worth listing. . . . The probabilities of winning, drawing and losing can perhaps be expressed in the form $(p^2, 2pq, q^2)$, and the above log-odds mean $\log(p^2/q^2)$. Judging by some statistics of E. T. O. Slater, p^2/q^2 is approximately 1·4 for the initial position in master chess. I believe that White's advantage is worth half a move or a fifth of a centre pawn, so a pawn is worth a factor of $1{\cdot}4^5 = 5$, and a queen $5^{9\frac{1}{2}} = 5{,}000{,}000$.'

(*c*) More convincing than the determination of the coefficients of a linear form, as an example of judgement, would be the determination of a functional form. The example of two bishops was mentioned.

(*d*) Very few human chess players discover new strategic principles for themselves. If we ask this of a machine we are asking for a level of creativity well above the average for humans.

(*e*) One advantage of an evaluation function expressed in terms of probabilities is that it could be worked in with estimates of the probabilities of the opponent's making various moves in any given position.

In 1956 a program was written at Los Alamos (Kister *et al.* 1957), capable of occasionally beating a beginner. The game was on a 6×6 board, and the analysis was exhaustive to two moves on each side. The evaluation was the simplest possible: a sum of material and mobility measures.

Bernstein *et al.* (1958a, b) wrote a program for the full 8×8 board, also

analysing two moves ahead, but with considerable move selection. As Newell, Shaw and Simon (1958) point out, this selectivity leads to a great increase in the complexity of the program, and so to a great slowing down *per move within* an analysis but that, in a deep analysis, this slowing down is far more than compensated because the number of variations to depth d goes up exponentially with d. If the average of the number of moves to be tried per position is *appreciably* decreased then the size of a deep tree is *enormously* decreased.

Newell, Shaw and Simon (1963, p. 47) mention *en passant* a hand simulation by F. Mosteller, and a Russian program for the BESM, but had little information about these two programs. Their own program avoids the use of a single evaluation on the grounds that humans do not seem to use one. They are especially concerned with the simulation of human thought, and they believe, as I do, that for a good chess program it is necessary to be so concerned. A number of goals are set up, and there is a move generator for each goal. Thus the evaluation function can be thought of as a vector. The goals that are listed are king safety, material balance, centre control, development, king-side attack, and promotion. Centre control and material balance are discussed in detail. Although the evaluation functions are not scalars, it is necessary to decide on an ordering for them in order that iterative minimaxing should be possible. One ordering suggested is lexicographic, the material balance being taken as the first component of the evaluation vector. It seems to me that any strict ordering of evaluation vectors effectively converts them into scalars. In lexicographic ordering, the components of a vector correspond to the digits of a scalar expressed relative to a large radix. The scheme is thus economical for the same reason that variable-length arithmetic sometimes is.

A recent reference is Newell and Simon (1965).

OUTLINE OF A FIVE-YEAR PLAN

The suggested plan can be split up into a number of parts, which I have listed here very roughly in the order in which they would be carried out. But this plan would be subject to revision from time to time. 'Progressive deepening' applies here just as much as to the analysis of a chess position! Note that the plan has not yet reached the software or hardware; it is in the form of so-called 'supporting underware'.

Five years seems a reasonable estimate for the project for a team of about three people working full-time. In the following breakdown of the work, I have not yet tried to allot these fifteen man-years.

4.1 Program a computer to play legal chess. This has of course been done several times before. It is not as easy as it sounds. A 'position' must be defined to include the information of whose move it is; whether the kings have moved, and, if not, whether the rooks have; whether $P \times P$ *en passant* is legal; how many times the position has occurred before with the same player to move (where here 'position' has its naive interpretation, and presumably no distinction is made between, for example, White's KR and QR); and how

many moves have been played since the last capture or pawn move. A move that gives check must be so described. A distinction must be made between the making of a move and its mere consideration. The legal-chess program should be efficient, since the speed of the *whole* chess program will be proportional to the speed of this one.

The lists that are worth keeping are those that are *cheap to up-date*. These should include a list of all squares occupied by White and by Black. Then, when we are considering White's moves, we do not need to run right through the 64 squares of the board to find the white men. (For the purpose of the whole program, for playing good chess, not merely legal, we should have lists of squares dominated by the two players.) Whether we are in check, and whether it is a double check, should be recorded from the opponent's previous move. If, for example, we are in double check, we know that we must move our king, in virtue of a theorem that happens to be true for chess although it would not necessarily be true if the rules were slightly different. If we are not in double check, then lists of our pinned pieces should be made before looking for legal moves, together with the *rays* along which they are pinned, and also whether each pin is *relative* or *absolute*, i.e., whether the ray is one along which the piece can or cannot move. There should be a *transient* list of our own pieces, from which we cross off each piece after we have listed all the legal moves we have made with it from the given position.

4.2. Collect chess principles from good books on chess, for example, Lasker (1932), Fine (1941), Horowitz and Mott-Smith (1960), Euwe and Kramer (1964-65), Suetin (1965), and de Groot (1965); by introspection and memory; and by dictating your chess thinking to a secretary or dictating machine. Each principle constitutes a potential goal. Here is a preliminary list.

Analyse 'forced' variations either to quiescence or to a point where the result of the game is obvious. A forcing move is one that gives little option to the opponent on his next move, and includes all checks. There are degrees of forcefulness, and the threshold on the forcefulness will vary according to the depth of the analysis from the current position. More precisely it should allow for the discussion of Appendix H concerning the meaning of an 'endpoint' in an analysis tree.

A special case of the above principle is the following deep thought: if you can mate on the move, do so.* Also, never miss a check, it might be mate,—a motto attributed in jest to the Hampstead Chess Club by J. F. O'Donovan. More generally, consider those moves that restrict the opponent's choice on the next move, even if this involves a sacrifice; but don't stalemate him except on purpose.

Material values: one conventional set is $P=1$, $N=B=3$, $R=5$, $Q=9$, but

* This could be extended: if you can mate in two moves, do so. For a complete analysis as far as your second move would be cheap on a large computer. In some positions an exhaustive analysis to n moves deep, for $n>2$, could be carried out by calling the subprogram mentioned in Section 2.1.

$B+B=7$. In particular, 'sacrifice' your queen if you get value for it. The advancing of a passed pawn can be regarded as the obtaining of extra material. For on the eighth rank a pawn is worth a queen. My own rule is that a passed pawn that cannot clearly be stopped is usually worth about five pawns on the seventh rank, three on the sixth, and one and a half on the fifth. A pair of double pawns $=1\frac{1}{2}$. 'The solution to every end-game is "pawn on" '—Doog. Swap pieces, but not pawns, when ahead in material, but don't let the position get completely blocked. See also Appendix F for values of pieces and squares.

In the opening, pawns can be arranged in decreasing order of value, according as they are Q or K pawns (central pawns), B pawns, N pawns, or R pawns. Lasker (1932, p. 107) gives the ratios 8: 6: 5: 2, but I should like to see a proper statistical survey of games to bear this out. Backward or isolated pawns are usually weak. If your opponent has them, blockade them with pieces. Two pawns, side by side, are strong because they cannot be blockaded. Try to fix a group of your opponent's pawns with a smaller number of your own. Avoid putting pawns on the same coloured squares as your lone bishop, especially if these pawns can be blockaded.

Increase your mobility, as compared with your opponent's. As discussed earlier, this includes more than is immediately apparent. To some degree it covers, for example, the advice to control space, and especially the centre, to get the pieces developed, and to move rooks to open and half-open files.

Attack the bases of your opponent's pawn chains, and defend the bases of your own.

Don't open up the position when you are behind in development.

Prevent your opponent from doing what you would do in his position. Decrease his mobility by advancing pawns. Do as you wouldn't be done by.

Exert pressure on the squares around the opponent's king.

The best strategy against a pawn storm on the wing is usually a thrust in the centre. If the centre is blocked and the kings are castled on opposite sides, then a pawn storm against the opponent's king is indicated. But if the pawns protecting the king are side by side, it is advisable to force one of them to advance before making the pawn storm.

When an attack fails, the side that was attacking is liable to be badly placed for the defence (as in Go), so don't attack without adequate reason.

Strengthen the weakest link in your position when defending, and attack the weakest link in your opponent's position when attacking. Move piece in worst plight. Exchange aggressive pieces when defending. Look for counter-attacks.

Preserve options, for example, develop at least one knight before both bishops in the opening, since the knights usually have fewer squares to go to.

Get pieces to aggressive positions if they can be held there; for example, knights on the fifth and sixth ranks on central or bishops' files, and rooks on the seventh rank for winning pawns and restricting the opponent's king.

Make your pieces co-operate. Co-operation means the achieving of sub-

goals by groups of pieces. For example, have a pawn protecting your advanced knight; connect your rooks by castling; double your rooks on open file or seventh rank. Avoid blocking your pawns with pieces.

Try to make double threats, as in many games.

Make each piece as efficient as possible without over-burdening it. The most important work should be done best if the costs are equal. For example, overprotect the centre, so that the pieces doing this work will have flexibility. (This is again a matter of mobility, but not according to the formal definition in terms of number of squares attacked.) Generally, flexibility is important. (See also the last sentence of this sub-section.)

In quiet positions, aim to accumulate small advantages. This should also be done in turbulent positions, but it is then more difficult to judge. (It needs a master's skill to carry out a complicated combination in order to win half a pawn!) If your small advantages are otherwise transient try to convert them into more permanent ones. Look harder for combinations, that is, lower the threshold of 'forcefulness', when your small transient advantages add up to as much as say 2/3 of a pawn, and also if your opponent has weaknesses such as cramped pieces, unprotected pieces, exposed king or queen, weak squares in front of king, queen and king in a line. The greater your non-material advantage, the lower the threshold of 'forcefulness' should be.

Standard won classes of end-games should be in store, since they provide endpoints on the analysis trees, and serve a similar function to the stored positions in Samuel's draughts program. A classification of mating positions and motifs serves a similar function (see Appendix G).

The following principles apply especially to the end-game.

An extra pawn is usually enough to win, if there are pawns on both the Q and K side of the board.

If each side has one bishop and these stand on squares of opposite colours, then an advantage of one pawn is usually not enough for a win.

A B is better than a N when there are pawns on both the Q and K side.

K and Q beat K; K and R beat K; K and two B's beat K; K, B and N beat K; K and two N's do not beat K; K and B beat K and 32 B's all on white squares!

If you are two pawns up (in the end-game) you can force a win by advancing them, since your opponent will have to give up a piece for them. This is a special case of the Law of Multiplication of Advantage.

K and P beat K in situations that could be completely formalised without great difficulty. For the present, note that it is useful to get the K in front of the P, and to get the 'opposition' (as in draughts).

K and Q usually beat K and R. The method, which is easy to express in language, but not to formalise, is to bring the K and Q close to the opponent's K and R, force them to the edge of the board, and, by threatening to win the R by pinning it, to force it away from the K. Then it can be forked or mate given.

A rook's pawn is stronger against a knight than other pawns are.

K, B, and RP beat lone K if the corner in front of the pawn is one the B can go to.

Put rooks behind your own passed pawns.

KUFTEG (king up for the end-game). For the king is worth four pawns in the end-game and should be used. If you are not yet sure where to use it, put it in the centre. It will then have less distance to go than it might have had; in other words the king, like the knight, has more flexibility when it is in the centre.*

4.3. Review the literature of chess programs to see if it contains any principles not already in the literature of chess.

4.4. Classify the various principles, in order that later formalisation should be more efficiently performed, and in order to find gaps.

4.5. See if the literature of learning theory, concept formation, pattern recognition, perception, and associative memory suggests new principles. If so, revise step 4.4. Don't devote five years to this step!

4.6. Try to express the various principles unambiguously and, where possible, quantitatively.

4.7. Devise a special-purpose chess language for expressing the principles as unambiguously as possible, but with not too large a vocabulary. (See Newell & Simon 1965, p. 62, for a preliminary list of chess vocabulary.) *Complete lack of ambiguity will not be essential if the application is man-machine communication, since the machine might be able to make syntactic transformations without full grasp of the semantics.* Ambiguous statements can be stimulating and concise, for *people.*

4.8. See if the terms of this language can be expressed in Algol or similar language.

4.9. Make a study of chess planning, to see whether it requires a language different from the expression of the chess principles. For example, one might say, 'Since the position is closed, I have time to get my knight round to Q5'. This will involve us in the study of the minimal syntax required for the convenient expression of chess principles and planning.

As an example of planning, consider the position in Fig. 1, from Lasker (1932, p. 24):

```
K . k . . . . .
P . . . . . .
N . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
```

FIG. 1. White to move. An example in which an efficient analysis is linguistic. Notation: lower case for Black.

* It is a universal principle of rational behaviour that flexibility has to be sacrificed when a decision is made, but a decision to *acquire* flexibility can be made. There is no contradiction here, since 'flexibility' can be achieved at various levels of a hierarchy.

A human might reason in the following manner. Could a machine do this?

'My opponent has only a king and I have a knight and pawn, so I can win only if I can promote the pawn (goal). In order to advance the pawn I must move my king to N7 or N8 (subgoal). This is prevented at present by the black king. He can continue to prevent it (his goal is to do so) by moving his king to B2 and back to B1 and so on (his plan). I must move my knight into such a position as to control his B2 when his king is at B1, or to control his B1 when he is at B2 (subsubgoal). Consult list of theorems about knights. Each time they move, the colour of their square changes. Ah, but each time the black king moves the colour of its square changes (inspiration). Therefore I can never succeed in my subsubgoal, and cannot win, if he continues with his plan.'

If we could put plans in order of merit, then plans, counterplans, counter-counterplans, etc., would form a *tree of plans*, and iterative minimaxing would apply to it.

4.10. Estimate the cost of having built-in facilities for making chess programming easier, on a general-purpose computer. Consider whether some of these facilities could be generalised so as to be useful for other kinds of problem-solving. For a review of more general problem-solving, see Newell and Simon (1961). Remember that the problems of hardware and software are inseparable. General problem-solving programs should not be divorced from general problem-solving machines.

4.11. Plan the effort so that it can be broken up into self-contained pieces, for clearer understanding and for parcelling out.

4.11.5. Work out how to organise a program so that positions that occur on an analysis tree, together with their analyses, are held in store for several moves in the course of a game, until the chance that they will occur in later analyses in the game becomes negligible (see also Section 4.23 below).

4.12. Write separate programs for combinations, for the openings (see Appendix J), for the middle-game, and for the end-game, just as separate books are written on these aspects of the game. What other aspects merit separate treatment?

4.13. Write out routines for winning several special end-games, without analysis trees, that is, by using procedures already known to chess players.

4.14 (continuing 4.8). Write a compiler in Algol, or in some other programming language, or in some modified form of an existing programming language, for the programs written in the chess language.

4.15. Assemble theoretical and experimental arguments for the evaluation of the relative values of the pieces, squares, and other features of the positions (see Appendix F). Can these be related to the changes in the log-odds of winning as against losing, or some such probabilistic measure? (see the discussion of Good 1959, in Section 3 above). Is there a general measure for the turbulence of a position? (see Appendix H). Chess programming and mathematical theory about chess should go hand in hand, but the programming will involve more work. Given the number of times that each player

controls each square, in a quiescent position, what is the player's expected advantage?

4.16. Work out further theory concerning optimal searches in trees. Is it possible to make a tentative evaluation of a chess or draughts *program* by calculating the statistical distribution of size of the analysis tree for given rules concerning the thresholds of the agitation at the endpoints? How should the thresholds for selecting endpoints on the trees vary with depth, with agitation, with the amounts of time left on the two faces of the chess clock? (see Appendix H). How should we estimate the value of storing previously analysed positions?

4.17. How should goals be generated? Each potential goal on a check-list can be considered, and rules worked out for assigning relative importance to them. I think we should aim at estimating the probability that we can achieve each goal, and the utility to us if we achieve it (see Appendix C). For this purpose, I think 'progressive deepening' is necessary (see Section 4.23 below).

4.18. How should the move-generators be organised, i.e., what *moves* should be ignored?

4.19. How should a variety of evaluation functions be handled, each appropriate to a different goal? Can evaluation functions be conveniently adjusted, during an analysis, instead of recomputing them in detail for each move considered?

4.20. Select about fifty quiescent chess positions, get a chess master to put them in order of merit for the player (that is, the side with the move), and then do the same with a variety of evaluation functions. Try to find an evaluation function, partly by hill-climbing, that puts the positions in nearly the same order as the chess master. If there are any bad discrepancies left over, think again and recheck with the master.

4.21. Correlate evaluation functions with those obtained by analysis plus iterative minimaxing, with the object of automatically improving the coefficients (see Samuel 1959, 1963).

4.22. Find evaluation functions that indicate whether a position cries out for startling moves, that is, moves that would be rejected, on a superficial analysis, by most of the goals. Without such evaluation functions, the machine will not be a genius.

4.23. Formalise the process of 'progressive deepening'. This is especially important for problem-solving in general. Example: the threshold at various depths of analysis should depend on how large the analysis tree is going to be. So we should first do a quick pilot analysis, for estimating the size of the tree, and then a more comprehensive analysis. There should be several stages in this process, since no simple method is likely to work. The difficulty is that a single branch of a tree can be greatly enlarged as a consequence of a slight change in the thresholds. The amount of the enlargement cannot be predicted. Hence pilot analysis might have to be applied to subtrees, as well as to the complete tree.*

* The device of holding in store positions that occurred on the analysis trees earlier in the game (see Section 4.11.5) will be relevant to pilot analyses.

Another application of progressive deepening is that ideas that occur at one stage can be used again at a later stage. For example, a move that was considered at one stage and seemed 'interesting' has a better chance of coming up for consideration at a later stage, whether or not it was sound at the earlier stage.

4.24. The relevance of progressive deepening to the formulation of sub-goals should be worked out, or at least worked on.

4.25. Play a few games and make improvements in the program as they suggest themselves.

4.26. 'Conversations' should be held with the computer in the special chess language (man-machine synergy). See what modifications in the chess language are required in order to answer new types of questions about chess. For example, we can ask the machine to count the number of distinct games in which White mates on his third move. (According to an analysis I did in about 1932, the answer is 367.) Another example would be retroactive analysis.

4.27. Try to find a transformation that converts an evaluation function into $p_W - p_L$ (or more generally into $p_W - \lambda p_L$), where p_W and p_L are the probabilities of winning and losing respectively. Try to do this even for turbulent positions (see Appendix H).

4.28. Investigate how the analysis should be modified in order to allow for the strengths, weaknesses, knowledge, habits, and style of a particular opponent.

4.29. Attempt to generalise the entire plan to a wider class of games. Here 4.26 will have been useful practice. Begin with randomised chess (see Appendix J), new and unusual kinds of pieces, and three-dimensional chess (see, for example, Good 1957). Many interesting variations on chess are known, such as Kriegspiel (which is not a game of perfect information), Buying Chess (Landau), Panzerspiel, Shotgun, chess with extra kings, chess without kings, chess on a torus, chess with mined squares, losing chess, and chess with a gradually increasing number of moves of each side. Randomised chess is special, since it is intended to replace ordinary chess, i.e., ultimately to be called 'chess'.

Note that, owing to limitations of human spatial visualisation, the machine might play better in three dimensions, relative to a man, than it would in two dimensions.

APPENDIX A. THE BOREL-VON NEUMANN THEORY OF GAMES, AND ITERATED MINIMAXING

In the 1920s, Borel and von Neumann founded a theory of games, which reached large book size in von Neumann and Morgenstern (1944). For the early history, and kudological remarks, see Borel, Fréchet and von Neumann (1953). In the zero-sum two-person game the two players independently choose two 'plays', i and j, which can be identified with the choice of a row and a column of a matrix, and the pay-off to the first player is a_{ij} and that to the second player is $-a_{ij}$. Neither player knows the other's play. The matrix (a_{ij}) is called the *pay-off matrix*. Chess is a special case of a zero-sum two-person game if we imagine that, according as the

first player loses, draws or wins, he collects -1, 0, or 1 from the second player. The play, *i*, is then to be interpreted as a set of rules which uniquely determine the player's move in each position, so that a play is not just a chess move, but is a single-valued function of positions taking values that are moves. For the selection of an optimal play, 'all' that is necessary is to work back from the end of the game, and deduce in turn whether each position is a theoretical win, draw, or loss for white. But the number of possible chess positions is about $10^{46\pm3}$ (see Appendix E) so an exhaustive analysis is virtually a physical impossibility. Perhaps the use of probability ideas in chess has the same kind of justification as its use in classical statistical mechanics!

The Borel-von Neumann theory can be applied to chess in a practical way only if the game is regarded as probabilistic, and then only in an exiguous manner. The theory shows, for example (what is obvious), that, if we can evaluate the position at all move-pairs from the current position, and if these evaluations are regarded as expected pay-offs, then we minimise our expected loss by playing the minimax move. This is the best move for us on the assumption that our opponent plays optimally, otherwise our best move is the one that maximises our expected gain after allowing for a probability distribution over all the moves that our opponent might make. When we do not assume that our opponent will play optimally, we are playing either psychological or trappy chess or both. All this can be extended to deeper analysis of a position, and leads, when we assume our opponent is playing well, to mini-maxi-mini-maxi-mini-maxing, etc., conveniently called *iterated minimaxing*, although the epithet 'iterated' is often omitted in discussions of chess and draughts. For psychological chess, a more appropriate term is (iterated) 'expectimaxing', a term that Michie (1966) applies to games against Nature, or against an opponent who moves randomly. Minimaxing is a special case of expectimaxing from the present point of view.

Since we associate $+1$ with a win for us, it would be more correct to use the term 'maximining' (and 'maxi-expecting') but 'minimaxing' is standard.

APPENDIX B. SOME COMMENTS ON SAMUEL'S DRAUGHTS (CHECKERS) PROGRAM

Samuel (1959) described a draughts program which is capable of sometimes beating near masters. His objective was to experiment with a learning program and he used an evaluation function whose coefficients could be varied by the program.* He resisted the temptation to store book variations, but he did store positions that occurred often enough in the experience of the machine. Presumably the effect of this 'rote' learning is that the program learns the openings to considerable depth, especially if it were pitted against a fixed player for a larger number of games in succession. (It can forget positions as well as learn them.) This device would be much less useful in chess since it has far more openings and not many of these extend almost to the end-game as they do in draughts.†

Samuel's paper was reprinted in 1963, together with the beginning of a game played in 1962 against Robert W. Nealey, who had been Connecticut champion, but was not at his top form. Unfortunately the end of the game was omitted in error. The position at the point where the record ceased is shown in Fig. 2 (i). The remainder of the game has been communicated to me by Dr Arthur Samuel, and is: 12-19, 32-27; 19-24, 27-23; 24-27, 22-18; 27-31, 18-9; 31-22, 9-5; 22-26, 23-19;

* It was not clear that these coefficients had reasonably well converged after several games. This makes one wonder how much of the evaluation function is really relevant to the success of the program.

† It would be interesting to have a typical frequency count $(n_0, n_1, \ldots)$, where n_r = number of stored positions in the draughts program at depth r, from the initial position.

26-22, 19-16; 22-18, 21-17; 18-23, 17-13; 2-6, 16-11; 7-16, 20-11; 23-19, White (Nealey) resigns.

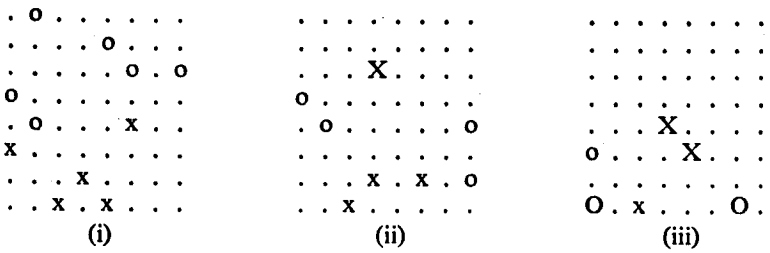


FIG. 2. Positions (i) and (ii) occurred in the game Machine v. Nealey, 1962; and position (iii) would have occurred if Nealey had played differently on his penultimate move. Notation: Black, crosses; White, noughts. Kings are in upper case.

In my opinion, White could have drawn the game if he had played 16-12 near the end in place of 16-11, which is a tactical blunder (see Fig. 2 (ii)). He would then have made a king which he would have been forced to exchange, but he could soon make another one. Black will be able to make a second king, but he will not be able to dig White's king out of the double corner, since White, in Fig. 2 (iii), can move his other king backwards and forwards while Black has his kings in the position shown, or further away. The interesting thing about this analysis, as mentioned in Section 1, is that it is not easy to see how it could be expressed in numerical terms.

In a letter of April 19, 1966, Dr Samuel remarks that, in a return mail match, Nealey won one game and drew five. Also that the machine played four games against each of W. F. Hellman, the world champion, and Derek Oldbury, a British champion, and that the machine lost all eight games. The machine, the IBM 7090, was held to five minutes per move at most, whereas the humans had no time limit. It seems reasonable to describe the machine as a near minor master, especially as the winning players required 70 to 80 moves to achieve winning positions. My guess is that the machine is very sound tactically but lacks judgement.

APPENDIX C. WHAT IS THE 'BEST MOVE'?

'Theoretically' we can do a complete analysis of any given position, π, and arrive at the endpoints of the tree, whose values (win, draw, or loss, for the machine) are known directly from the rules of chess. Then we can backtrack, by iterated minimaxing, to get the values of all the positions that can be reached from π in one 'half-move', i.e., one move by the player whose turn it is. In this way we can in principle determine the moves that give nothing away, and, from an over-logical point of view, we could describe these moves as 'equal best'. Then again, we could instead define the best move (or equal best), if π is theoretically won, as a move that forces checkmate in the minimum number of moves against any play however good. The best move in a lost position would analogously be defined as one that delays mate for as long as possible. If π is a theoretical draw, then in much the same spirit we could define the best move as one that delays for as long as possible an indisputable draw (assuming the 50-move drawing rule to be mandatory). For since, at the moment, we are assuming the machine to be a perfect player, we should like the opponent to have as many opportunities as possible for making a mistake. Or it might be a little more logical to maximise, not the number of moves to an indisputable draw, but the product of the number of losing choices that the opponent would have at each of his moves. For this is the total number of ways he has of

going wrong. It is a little more realistic simply to maximise the probability that he will go wrong, and this brings us close to a practical definition of 'a best move'.

Suppose that, as a result of each of the moves the machine can make in position π, we can estimate the (superficial) probabilities p_W, p_D, p_L, that we (the machine) will win, draw or lose. For a given state of the tournament, prize-list, etc., these possibilities will have various utilities, u_W, u_D, u_L, for us. The expected utility of the move will therefore be of the form $p_W u_W + p_D u_D + p_L u_L$. But since $p_W + p_D + p_L = 1$, this expected utility is of the form $\alpha p_W - \beta p_L + \gamma$. Since we are concerned only in finding the best move, we can drop the constant γ, and also divide out by α. Hence the expected utilities, up to an irrelevant linear transformation, are of the form $p_W - \lambda p_L$. A similar argument can be applied by the opponent, so that, if for the moment we regard the probabilities as objective, we can infer that $\lambda = 1$ if the game is 'zero-sum'. So it is fairly reasonable to take the expected utility as $p_W - p_L$; but $p_W - \lambda p_L$ is more reliable.

If our evaluation functions enable us to infer the value of $p_W - \lambda p_L$, then we could 'in principle' backtrack and find the best moves in the sense of expected utility. If we were dealing with a game, like Hex, in which there are no draws, then the best move in the sense of the principle of rationality would also be the best in the sense of maximising the probability of winning.

There is a slight over-simplification in the above discussion since, in many tournaments, there is more utility in winning quickly than slowly. It preserves your energies for the next game. I have ignored this effect in this paper.

APPENDIX D. THE NUMBER OF POSSIBLE GAMES OF CHESS

If fifty moves are played on each side without any captures or pawn moves, then, in some sets of rules, the game is drawn, whereas, in other sets, the draw must be claimed. If the rule is taken as mandatory, then no game can last more than 6000 moves on each side. Also, in any position, even if all the pawns have been promoted to queenhood, the number of possible moves cannot exceed 321. Therefore the number of possible games is less than $321^{12\,000} < 10^{30\,000}$. (This inequality was given by Good 1939. With a little more work, the upper bound can be lowered somewhat.)

If we restrict our attention to reasonable games, or to master games, we get a much lower estimate. The method of estimation is similar to one used in certain Monte Carlo estimation problems having no reference to games (see Good 1954). If a_n is the expected number of moves, at a given standard of play, at the nth ordinal position, counting the initial position as the first, then the total expected number of games of not more than $2m$ moves on each side, for m less than say 200, whose moves are all of the approved standard, is about

$$\sum_{n=1}^{2m} a_1 a_2 \ldots a_n (1-\epsilon_1)(1-\epsilon_2)\ldots(1-\epsilon_{n-1})\epsilon_n,$$

where ϵ_n is the probability that the game would terminate immediately after the nth move.

de Groot (1965, p. 25) mentions the analysis of one game in which the average number of good moves per position did not exceed 2, but he does not state what the average was. The analysis of several games would be required in order to obtain reliable estimates of the a_i's($i = 1, 2, 3, \ldots$). For $n > 15$, an adequate approximation to ϵ_n is 1/40, as far as the present analysis is concerned.

Another approach is to consider how many opening lines are recorded in Evans (1965). There are about ten thousand lines given to an average depth of about 12 moves, so the average number of moves considered in each position is about

$$10\,000^{1/24} = 1{\cdot}48.$$

These are almost entirely lines that have been played or analysed by good players. It seems a reasonable guess that

$$1{\cdot}6<(1-\epsilon_n)a_n<1{\cdot}9$$

for most values of n, if extrapolation from the openings to the rest of the game is justifiable. (It is interesting to observe that, if 1·75 is the correct average in the opening, then *Modern Chess Openings* ought to run to fifty volumes, that is, only a fiftieth of equally good lines are at present recorded. This might give some measure of the conventionality of 'the book'.)

If this guess is right, then the number of games of not more than m moves on each side is roughly

$$\frac{1}{40}\cdot\frac{\mu^{2m+1}-1}{\mu-1}$$

where $1{\cdot}6<\mu<1{\cdot}9$, that is, the number lies between $1{\cdot}6^{2m+1}/24$ and $1{\cdot}9^{2m+1}/70$ These bounds are tabulated approximately in Table 1.

TABLE 1

The number of good games of not more than m moves on each side

m	10	20	30	40	50	100	200
$\log_{10}$ (lower bound)	3	7	11	15	19	39½	80½
$\log_{10}$ (upper bound)	4	9½	15	20½	26	54	110

It is interesting to note that, although some 99 per cent of games terminate before White's hundredth move, nevertheless the number of possible well-played games longer than this is far greater than the number that are shorter.

The number of good 'lines', that is, games not necessarily terminated, up to Black's mth move, is about $(\mu^{2m+1}-1)/(\mu-1)$ and is bounded by about $1{\cdot}6^{2m+1}$ and $1{\cdot}9^{2m+1}$. These bounds are tabulated in Table 2.

TABLE 2

The number of good 'lines' up to Black's mth move

m	10	20	30	40	50	100	200
$\log_{10}$ (lower bound)	4½	8½	12½	17	21	41	82
$\log_{10}$ (upper bound)	6	11½	17	22½	28	56	112

The above table can be inverted. For example, we can infer from it that a catalogue of 10^{12} good lines would extend to between 21 and 29 moves on each side, or say 25 moves, give or take a few moves. Moreover, if the machine had the white pieces, and was satisfied to adopt a fixed repertoire, 'in principle' it could get to about this depth with a catalogue of only a million lines (this assumes the square root law mentioned in Appendix J). As soon as the opponent played a slightly inferior move, the machine would have to begin exercising intelligence.

APPENDIX E. THE NUMBER OF POSSIBLE CHESS POSITIONS

Shannon (1950, p. 260) states that the number of possible chess positions is of the general order of $64!/(32!8!^2 2!^6)$ or roughly 10^{43}. The formula indicates that he

was assuming that no pawn had been promoted. In Good (1951), I calculated that the number of positions in which no pawn has been promoted, and there are no doubled pawns, is less than 2×10^{39}. The number of positions in which no capture has occurred is about 10^{32}. Allowing for all possibilities the number is less than $64.63\sum_{r=0}^{30}\binom{62}{r}10^r$ which is about 2×10^{50}, but if the number of moves since the last capture or pawn move is regarded as part of the position, the upper bound is 10^{52}. I should say that 10^{46} is correct for the total, within a factor of a thousand. But most of these positions would be fabulously improbable. A master chess player would be happy if he knew what the best move was in 99·9 per cent of the positions, with nearly level material, weighted with their probabilities of occurring, that would occur in master chess, without blunders. Judging by the tables in Appendix D, and allowing for the possibility of repetitions of positions in the end-game, it may be that this number would not be more than about 10^{24}. Conceivably a good estimate of the total number could be made if a very comprehensive categorisation of chess positions were first produced, in which positions would fall into clumps of trillions of positions each. Another approach would be to choose a large number of dispositions for the pieces, without pawns, and, for each of them estimate the number of ways of distributing the pawns legally—in effect a method of stratified sampling.

To make a rough comparison of the number of possible chess positions with that of draughts positions, the very crude upper bounds, 13^{64} and 5^{32} or 3^{32}, suggest that the former is about the cube or fourth power of the latter.

APPENDIX F. THE VALUE OF THE PIECES AND SQUARES

A satisfactory theory of the values of positions should include a theory for the values of the pieces as a special case. So it is worth while to look for a theory that gives reasonable values to the pieces. Then we could try to generalise the theory.

The values of the pieces P, N, B, R and Q, have been found by experience to be approximately proportional to 1, 3, 3, 5 and 9, and a king is worth about 4 in the end-game. These values vary with the position and with the number of pieces on the board. For example, two knights are worth less than rook when the only other pieces on the board are two kings, in so far as two knights and a king cannot force mate against a lone king. But the values given are applicable rather widely. For example, two knights are about the equal of six pawns on the second rank even when the kings are removed, as I have found by experiment.

A theoretical attempt to evaluate the pieces was made by H. M. Taylor in 1876, reported by Coxeter (1940, pp. 162-165). The value of a piece is taken as proportional to the average number of squares controlled, averaged over all 64 positions of the piece on the board. This argument leads to the relative values of N, B, R and Q: 3, 5, 8 and 13. Coxeter (or Taylor) goes on to modify the argument by asking instead for the probability of 'safely' giving check, that is, without being *en prise* to the king, if the piece and king are both placed on the board at random. This gives the ratios 12, 14, 22 and 40. These values are good, but this modification of the argument is artificial. For the piece values seem to be valid even for chess without kings, as I have found from some limited experiments, such as the one just mentioned. In fact they are probably more valid when there are no kings. Consider, for example, a piece, which I shall call the Galactic Emperor, that controls all the squares on the board. When the kings are on, it is worth at least seven queens, but, measured by squares dominated, it is worth only about three queens, on an open board. Therefore I believe that a theory that allows for the presence of the kings is too difficult to start with, and that it is more sensible to try first to obtain a theory that arrives at the known values without reference to checking.

In about 1936 (unpublished) I tried to make some allowance for the number of pieces on the board:

Let p be the probability that a square of the board is occupied. I assumed that the value of a piece is proportional to the expected number of squares dominated, averaged for all positions of the piece, and obtained the following values for the knight, bishop and rook.

$$E_N(p)=5{\cdot}25 \text{ (independent of } p),$$
$$E_B(p)=(140-196p+182p^2-112p^3+44p^4-10p^5+p^6)/16,$$
$$E_R(p)=\frac{4}{p}-\frac{1}{2p}\left\{\frac{1-(1-p)^8}{p}\right\}.$$

The expected number of squares dominated by the queen is $E_Q(p)=E_R(p)+E_B(p)$. But we should allow also for the fact that rook and bishop have overlapping fields of influence. For example, a rook on Q5 and a bishop on KB4, on an open board, dominate only 24 squares instead of 27, and so lose about one ninth of their value. This gives a partial explanation of why a queen is worth more than a rook plus a bishop. A similar argument explains why two bishops are worth more than a bishop and knight, or than two knights. It is usually better to control two squares than to control one square twice, not allowing for the different values of squares.

Table 3 gives the numerical values of $E_R(p)$, etc., for a few values of p.

TABLE 3

Values of $E_N(p)$, etc.

p	1	$\frac{1}{2}$	1/3	1/4	0
N	5·25	5·25	5·25	5·25	5·25
B	3·07	4·7	5·72	6·35	8·75
R	3·5	6	7·5	9	14

The case $p=1$ cannot occur in ordinary chess, but it is certainly reasonable that a knight would be worth more than a rook if all 64 squares were occupied. The case $p=\frac{1}{2}$ is not expected to apply for the first several moves of a game, when we know roughly where the pieces stand. The initial position can be examined in its own right, as indeed can any given position. But it would be too short-sighted to say that rooks, bishops and queens have no value in the initial position, merely because they cannot move. It would be more reasonable to make some guess concerning the probability that each piece will occupy the various squares after any given number of moves, and then to use these probabilities as weights in a weighted average of the numbers of squares dominated. Moreover we should discount the future in some way, for example, by giving further weights of μ, μ^2, μ^3, . . ., to positions 1, 2, 3, . . . moves ahead, where μ is some positive number less than unity.

The case $p=1/3$, which is typical in the middle-game, gives values that are generally regarded as good average values of the pieces. It would be interesting to try to verify the remainder of Table 3 experimentally.

To make the theory more accurate we may attach a value to each square, making the central squares more valuable than the others. To do this we can use a similar principle to the one used for pieces, that is, we can find the average number of squares dominated from the square when various pieces are placed there. These can be taken as 1Q, 2Rs, 1B (only one colour!), 2Ns, but not 8Ps since not every pawn has a reasonable chance of getting to the given square. It seems reasonable

to assume $1\frac{1}{2}$ pawns. I found the following values for the squares by this method. The squares are labelled as in Fig. 3.

	D	E	F	I
	B	C	E	H
	A	B	D	G

FIG. 3. Labelling of squares. The labelling is centrosymmetrical, apart from the edges.

A: $87-84p+44p^2-8p^3$.

B: $83-82p+55p^2-20p^3+3p^4$.

C: $83-92p+80p^2-40p^3+8p^4$.

D: $75-90p+95p^2-50p^3+18p^4-3p^5$.

E: $75-100p+110p^2-70p^3+23p^4-15p^5$.

F: $71-120p+160p^2-120p^3+48p^4-8p^5$.

G: $65-108p+130p^2-110p^3+63p^4-21p^5+3p^6$.

H: $65-118p+155p^2-130p^3+86p^4-21p^5+3p^6$.

I: $63-138p+205p^2-180p^3+93p^4-26p^5+3p^6$.

We can use these values of the squares to improve the estimates for the values of the pieces. We could then use these improved estimates to improve the estimates for the squares and so on. There will always be an error because of the assumption that the density of p is the same over the whole board, and also because we can *choose*, for example, not to keep knights at the edge of the board, as a general rule. The chance that a knight is on the edge, after the opening, is much less than 28/64, if the players are any good!

I did this algebra before there were any electronic computers. It will now be easy to do the arithmetic.

If we wish to allow for the existence of kings we should give extra weight to the squares in their vicinities, or in some other way. We might, for example, compute a 'checking value' for each piece as the probability of giving check without being *en prise* to the king, and then take a linear combination of the checking value and the square-domination value.

The values of the pawns require a different theory since (i) they can be promoted, and (ii) they move differently from the way they capture.

It should be considered whether the values of the pieces can be related to the probability of winning. Should the value be taken as the improvement in the log-odds of winning as against losing?

Theories of evaluation of pieces could be tested by inventing new pieces; we are not inevitably tied to the small sample of distinct kinds of piece that occur in ordinary chess. An example of this test, was given above in connection with the Galactic Emperor, although perhaps this piece is too extreme for most purposes.

APPENDIX G. CATEGORISATION OF MATING POSITIONS AND CATALOGUE OF FREQUENT MATING MOTIFS

When attacking the king, human players find it useful to be able to recognise the possibility of mating motifs and mating positions. To have a repertoire of these

things is helpful in goal formation and in the pollarding or pruning of analysis trees.

For mating motifs, a useful reference is Vukovic (1965), and there are many other books that deal with this topic. Here I should like to mention a categorisation of mating positions.

(a) *Three-filer.* An example is a mate with two rooks, the opponent's king being on the edge of the board. It is natural to think of the three squares off the board as forming an imaginary file. Another example is mate with a rook and king against a king on the edge, or a mate on the back line, when the opponent's king has three pawns in front of it.

(b) *T and tunnel.* For example, the final position in the game 1 P-K4, P-K4; 2 Q-R5, K-K2; 3 Q×KP mate.

(c) *Cross diagonals and L.* For example,

```
. . . N . . R .        . . . . . . . K
. . . . p k . P        . . . B . . Q .
. . . . . p . .        . . . . . k . .
                       . . . . . . . .
```

(d) *Parallel diagonals and two stops.* For example,

```
. . k r . . . .
. p . . . . . .
NN. . . . . .
```

With a little artificiality, most mating positions can be put into one or more of these categories. If a machine could do this it would really have arrived.

APPENDIX H. QUIESCENCE, TURBULENCE, AND AGITATION

Positions for which there is no move forceful or purposeful enough to be worth analysing were independently described as 'quiescent' by Shannon (1950) and by Good (1950b). Turing (1953) called such positions 'dead'. One objection to this term is that a 'dead draw' is a standard expression among chess players. Also it is useful to be able to refer to 'degrees of quiescence', whereas 'degrees of deadness' sounds facetious, in spite of the expression 'dead as a door-nail' and of the notorious variability of deadlines. A good name for the degree of quiescence is simply the 'quiescence' of the position, or inversely one might refer to the 'turbulence'. A quiescent position is one of low turbulence, a combinative position is one of high turbulence. It is possible to define turbulence with respect to goals other than material balance, such as control of the centre (cf. Newell *et al.* 1958). The position illustrated in Fig. 4 is of high turbulence with respect to material balance.

```
. . . . . . . .
. . k n R . . .
. . . r Q b B .
p p p b q n p p
P P P R N N P P
. . . p p r B .
. . . P P P . .
. . . . . K . .
```

FIG. 4. A position of high turbulence!

Before considering a more precise definition of 'turbulence' we discuss an allied notion called 'agitation'.

If, in the analysis of a position π_0 we reach a position π, the turbulence of π is not quite an adequate criterion for deciding whether to suspend further analysis. If π were obviously won we would suspend its analysis even if it were turbulent. We need some measure, called say 'agitation', that measures turbulence and also takes into account whether the position is obviously won, drawn, or lost. If we can measure the agitation of π, then we would set a threshold on this agitation multiplied by $P(\pi|\pi_0)$, the probability of reaching π from the current position π_0. (Notice that, when we estimate this probability, we are in part, estimating the probability of our own future actions, not just those of our opponent. In my opinion, the same thing happens in the definition of a 'decision'. See Good 1964.) If this threshold is not reached, we would almost suspend analysis of π. I say 'almost' because there are a few points to attend to before finally classifying π as an endpoint on the analysis tree of π_0. We should first (i) check whether π is a position known definitely to be won, drawn, or lost; (ii) if π is favourable to us, we should see if we have a move that prevents check; (iii) if π is unfavourable to us, we should see if we have a check; (iv) if π seems to be an obvious draw, we should carry out both the tests (ii) and (iii).

How then should 'agitation' be measured? It will be recalled from Appendix C that the utility of position π, if we reach it, is of the form $P(W|\pi)-\lambda P(L|\pi)$, where $P(W|\pi)$ and $P(L|\pi)$ are, in some sense, the probabilities of a win or loss, given π. If we decide to treat π as an endpoint, then these probabilities are 'superficial' and $P(\mathrm{W}|\pi)-\lambda P(\mathrm{L}|\pi)$ may reasonably be described as the 'superficial expectation of the utility' or, for short, as the 'superficial utility'. If we decide to suspend the analysis of π, that is, to regard it as an endpoint, it should be because we believe that a deeper estimate of the utility of π, obtained say by spending a dollar on its analysis, is unlikely to make much difference. We must therefore have some method of guessing the expected value of $|U(\pi|\$)-U(\pi)|$, where $U(\pi)$ is the superficial utility of π, and $U(\pi|\$)$ is the expected utility of π in the light of a dollar's worth of analysis, \$, or of course some other unit. The *agitation* $A(\pi)=A(\pi|\$)$, of π is the guessed expected value of $|U(\pi|\$)-U(\pi)|$, and the decision whether to treat π as an endpoint in the analysis of π_0 is to be made by setting a threshold on $A(\pi)P(\pi|\pi_0)$. Part of the five-year plan is to find an evaluation function that estimates $U(\pi)$, and another one that estimates $A(\pi|\$)$. We need evaluation functions even for turbulent positions.

A well-known issue in the theory of rationality is whether the distribution of a utility is relevant except in regard to its expectation. My own belief, shared by most 'rationalists', is that only the expectation is relevant, but this is on the assumption that all the relevant calculations have been completed. In complicated problems, like mathematics and chess, this assumption is unrealistic, and superficial probabilities and expectations have to be used (cf. Good 1962b). The guessed expected value of $|U(\pi|\$)-U(\pi)|$ is roughly proportional to the standard deviation of the superficial subjective distribution of the true utility of π. The true utility of π is of course either 1 or $-\lambda$, or $\frac{1}{2}(1-\lambda)$, if both we and our opponent are perfectly rational.

I believe that this notion of the agitation of a position is applicable to almost every decision in practical life, since it is very rare that we can complete our reasoning in a practical problem. Instead of using the axiom of probability that $P(E|H)=P(F|H)$ when E and F are logically equivalent, we are forced to use the weaker one that these probabilities are equal when we have *proved* that E is equivalent to F (Good 1950a, p. 49.) Both humans and machines must often use imperfect logic.

It seems reasonable to define the turbulence, $\tau(\pi|G)$, of π with respect to some goal G, as the superficial expectation, or guessed value, of $|L(G|\pi, \$)-L(G|\pi)|$, which equals $|W(G:\$|\pi)|$, where L denotes log-odds, W denotes 'weight of evidence', and the colon denotes 'provided by'. But it seems that decisions should depend

more on agitation than on turbulence. The utilities should allow for all goals that we are prepared to consider. Agitation with respect to a goal can also be defined by interpreting U appropriately.

If these ideas could be properly worked out, they would be applicable to the question of finding the value of storing previously analysed positions, as in Samuel's draughts program.

At the end of Section 1, we mentioned the problem of deciding what moves to discard 'without analysis'. I think the right approach to this question is to try the move, and estimate $|U(\pi')-U(\pi)|P(\pi|\pi_0)$ where π' is the position reached after that move is played, and the utility $U(\pi')$ is estimated on the (false) assumption that the player retains the move. If this estimate is small enough, then the move can be treated as of no importance. In fact, if this estimate is small, then $P(\pi'|\pi)$ would be small, since a free move is usually quite advantageous. An exception is when the player is *Zugzwang*, and also in some end-game positions when no move is any good.

APPENDIX J. HOW SHOULD THE OPENINGS BE PROGRAMMED? SHOULD CHESS BE REPLACED BY RANDOMISED CHESS?

One obvious suggestion for handling many of the openings is to store nearly the whole of *Modern Chess Openings*. Then the machine will not go badly wrong until the opponent leaves the book. Moreover, some of the published lines would be clear wins for the machine, whereas those that were known to be bad for the machine could be avoided. In correspondence chess, this device would put the machine at a disadvantage, but in 'over the board' chess, the machine would sometimes win without any 'intellectual effort'. This would indeed be a hollow victory.*

Chess was presumably invented as a test of a kind of intelligence, and it is no more reasonable that the machine should be allowed to win by throwing the whole book at its opponent than that humans should be allowed to win tournaments by the learning of innumerable lines. Both for machine programming, and for the health of chess in general, the present rules of the game should undoubtedly be modified, and so-called 'randomised chess' should be adopted by the International Chess Federation, the F.I.D.E. (see, for example, Good 1952-55, 1965, p. 35).

In randomised chess the white pieces on the back line are randomly permuted before the game begins, but if the two bishops turn out to be on squares of the same colour the one on the right is interchanged with the piece on its left. This gives rise to 1440 essentially distinct positions if the black pieces are set up by mirror reflection. Castling can be defined by the rule that the rook is moved up to the king and the king hopped over it. Randomised chess has exactly the same seriousness as ordinary chess, and the basic principles, such as the need to control the centre, are the same. The theory of the openings is genuine theory and not largely practice and fashion as in ordinary chess.

But since it is too much to hope that the F.I.D.E. will be rational enough to organise this change in the face of opposition from vested interests, let us consider what we can do other than swallowing the whole book.

The chess openings form a very large tree: there are about 150 000 moves in *Modern Chess Openings*, consisting of some 10 000 lines. But a player with the white pieces, and a fixed repertoire, need learn only about 100 lines, the square root of the number in the book, and another 100 lines for when he has the black pieces. The argument for this, which is not at all rigorous, is that the number of lines in the book is roughly k^n, where k is the number of choices in each position, and n is the length of the line; whereas a repertoire for White need have only about $k^{n/2}$ lines, since he can make a fixed choice in each book position. The argument

* It would not be hollow if it provoked the F.I.D.E. to go over to Randomised Chess.

is not rigorous because k is a decreasing function of the number of moves from the initial position, since the book is intended to be a book about the *openings*.

At any rate, one way that a computer could be used to help a chess player would be in the selection of an opening repertoire for him. (A repertoire is a Borel-von Neumann pure 'strategy' or 'play', but is incomplete.) If the repertoire were selected randomly, then the player's opponents would not be in a position for some time to take advantage of his published games in order to prepare variations against him. Likewise the machine could select a repertoire for say its next ten games, It would then be able to start a game quickly 'over the board', and perhaps finish by correspondence.

The problem of selecting an opening repertoire or subtree can be framed in several ways. For example, we could try to find a repertoire having minimum storage requirements (which would be especially useful for human players), or one with a preference for open games. It might well turn out that a machine would play a better combinative than strategic game in comparison with humans.

APPENDIX K. SYNERGY, A POINT OF TERMINOLOGY

Close interaction between men and machines is sometimes called 'man-machine symbiosis' and sometimes 'man-machine interaction'. The first term is inappropriate since, in correct usage, it refers to purely biological interaction. The second term does not convey the notion of the closeness of thé interaction. Fortunately there is an English word already available, it is 'synergy'. In my opinion the 1970s will be remembered as the age of synergetics.

REFERENCES

Bernstein, A. *et al.* (1958a). A chess-playing program for the IBM 704 computer. *Proc. Western Jt Computer Conf.* (*WJCC*), 157-159.

Bernstein, A., & Roberts, M. de V. (1958b). Computer vs. chess-player. *Scient. American*, **198**, 96-105.

Borel, E., Fréchet, M., & von Neumann, J. (1953). Discussion of the early history of the theory of games, with special reference to the minimax theorem. *Econometrica*, **21**, 97-127.

Coxeter, H. S. M. (1940). *Mathematical Recreations and Essays*, from the original by W. W. Rouse Ball. London: Macmillan.

Euwe, M., & Kramer, H. (1964-65). *The Middle Game*, two volumes. London: Bell.

Evans, Larry (1965). *Modern Chess Openings* (Griffith and White), 10th edn. W. Korn (ed.). London: Pitman.

Fine, R. (1941). *Basic Chess Endings*. Philadelphia: David McKay; London: Bell.

Golombek, H. (1966). Luck in chess. *Observer Weekend Review*, January 23, p. 31. London.

Good, I. J. (1939). Mathematics and chess, *Eureka*, **1**, 9-11.

Good, I. J. (1949). The number of individuals in a casade process, *Proc. Cambridge Philosophical Soc.*, **45**, 360-363.

Good, I. J. (1950a). *Probability and the Weighing of Evidence*. London: Griffin; New York: Hafner.

Good, I. J. (1950b). Contribution to the discussion on Slater (1950), p. 199.

Good, I. J. (1951). Private communication to Mr Leon Jacobson.

Good, I. J. (1952/55). Notes on Randomised Chess in *Chess*, **17**, 161, 1952; **20**, 203, 1955; *Brit. Chess Mag.*, **73**, 213-214, 1953.
Good, I. J. (1954). Contribution to the discussion of a paper by J. M. Hammersley and K. W. Morton. *J. Roy. Stat. Soc. B*, **16**, 68.
Good, I. J. (1957). Letter in *Chess*, **22**, 293-294, containing misprints, and concerning a form of three-dimensional chess played on a $4 \times 4 \times 6$ 'board'.
Good, I. J. (1959). Could a machine make probability judgments? *Computers and Automation*, **8**, 14-16, 24-26.
Good, I. J. (1962a). Botryological speculations, *The Scientist Speculates*, I. J. Good, A. J. Mayne, J. Maynard Smith (eds.), 120-132. London: Heinemann; New York: Basic Books.
Good, I. J. (1962b). How rational should a manager be? *Management Sc.* **8**, 383-393; and in *Executive Readings in Management Science*, M. K. Starr (ed.). New York: Macmillan (1965).
Good, I. J. (1964). Measurement of decisions, *New Perspectives in Organisation Research*, pp. 391-404. W. W. Cooper, J. Leavitt, and M. W. Shelly II (eds.). New York: Wiley.
Good, I. J. (1965a). Speculations concerning the first ultraintelligent machine, *Advances in Computers*, Vol. 6, pp. 31-88. F. L. Alt and M. Rubinoff (eds.). New York and London: Academic Press.
Good, I. J. (1965b). 'The mystery of Go.' *New Scientist*, **25**, 172-174.
Groot, Adriaan D. de (1965). *Thought and Choice in Chess*, G. W. Baylor (ed.). The Hague and Paris: Mouton. A translation, with additions, of a Dutch version of 1946.
Horowitz, I. A., & Mott-Smith, G. (1960). *Point Count Chess*. New York: Simon and Schuster.
Kister, J., Stein, P., Ulam, S., Walden, W., & Wells, M. (1957). Experiments in chess. *J. Assn Comput. Mach.*, **4**, 174-177.
Lasker, Edward (1951). *Modern Chess Strategy*. London: Bell.
Lasker, Emanuel (1932). *Lasker's Manual of Chess*, 2nd ed. London: Constable; Dover reprint, 1960.
Luce, R. D., & Raiffa, H. (1957). *Games and Decisions*. New York: Wiley.
Macdonald, D. K. C. (1950). Contribution to the discussion on Slater (1950), p. 198.
Maynard Smith, J., & Michie, D. (1961). Machines that play games. *New Scientist*, **12**, 367-369.
McCulloch, W. (1965). *Embodiments of Mind*. Cambridge, Massachusetts: M.I.T. Press.
Michie, D. (1966). 'Game-playing and game-learning automata', with an appendix by J. Maynard Smith; in *Advances in Programming and Non-Numerical Computation*, pp. 183-200. Oxford, etc.: Pergamon Press.
Neumann, J. von, & Morgenstern, O. (1944-47). *The Theory of Games and Economic Behaviour*. Princeton: University Press.
Newell, A., Shaw, J. C., & Simon, H. A. (1958, 1963). Chess-playing programs and the problem of complexity. *IBM J. Res. Dev.* **2**, 320-335, 1958. Reprinted in *Computers and Thought* (see Samuel 1963), 39-70.
Newell, A., & Simon, H. A. (1961). Computer simulation of human thinking. *Science*, **134**, 2011-2017.
Newell, A., & Simon, H. A. (1965). An example of human chess play in the light of chess playing programs. *Progress in Biocybernetics*, pp. 19-65. Amsterdam, London, New York: Elsevier.
Philidor, F. A. D. (1749). *L'Analyse des Échecs*. London.
Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, **3**, 210-229.

Samuel, A. L. (1963). Reprint of Samuel (1959), with an additional appendix. In *Computers and Thought*, E. A. Feigenbaum and J. Feldman (eds.), pp. 71-105. New York, etc.: McGraw-Hill.

Shannon, C. E. (1950). Programming a computer for playing chess. *Phil. Mag. (London) 7th Ser.*, **41**, 256-275.

Slater, Eliot (1950). Statistics for the chess computer and the factor of mobility. *Symposium on Information Theory*, pp. 150-152. London: Ministry of Supply.

Suetin, A. S. (1965). *Modern Chess Opening Theory* (tr. from the Russian by D. J. Richards, ed. by P. H. Clarke. Oxford, etc.: Pergamon; date of the original Russian suppressed).

Turing, A. M. (1953). Digital computers applied to games, *Faster than Thought*, B. V. Bowden (ed.), pp. 286-295. London: Pitman.

Vukovic, V. (1965). *The Art of Attack in Chess* (tr. by A. F. Bottrall, ed. by P. M. Clarke. Oxford, etc.: Pergamon.

8

NEW DEVELOPMENTS OF THE GRAPH TRAVERSER

JAMES DORAN
EXPERIMENTAL PROGRAMMING UNIT
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

INTRODUCTION

This paper describes some recent experiments with a computer program which is capable of useful, or at least interesting, application to a number of different problems. The program, the Graph Traverser, has been described in detail in a previous paper (Doran & Michie 1966). However, we shall here need to view the basic algorithm from a rather more general standpoint, corresponding to an actual extension in the flexibility of the program, so that a restatement of what the program can do is desirable.

The Graph Traverser, which is written in Elliott 4100 Algol, is potentially applicable to problem situations which can be idealised in the following way (see for comparison Newell and Ernst 1965). There is given a set of 'states', which are connected by a set of 'transformations', or, as I shall call them, 'operators'. An operator will be applicable to some, but not necessarily all, of the states and two distinct operators applied to either the same or distinct states may each give the same state as end-product. Most of the concepts to be used here which are related to the use of operators were discussed in a paper by Michie (1967).

This type of problem situation is represented in Fig. 1 by a graph (in the mathematical sense) to which have been added various labels. In this representation states correspond to nodes of the graph, and operators to labelled arcs—*a*, *b*, *c* in this quite arbitrary case. Notice that associated with each node (or state) is a triad of integers. These represent the way in which the

program handles the 'structure' that is typically associated with a state, for with each node it holds an integer matrix, the dimensions of which will depend upon the particular problem. It is this structure which embodies the additional information which the program uses to carry out a heuristically directed search for a solution rather than a systematic search.

What 'idealised' problems can be posed in this context? It is convenient to recognise two types (for a fuller classification see Doran & Michie 1966). These we may call informally 'state-search' problems and 'path-search' problems. In a state-search problem we wish to find a state with some particular property. For example, and this will be relevant later on, we might seek the state for which some associated cost was smallest. In Fig. 1 we might seek

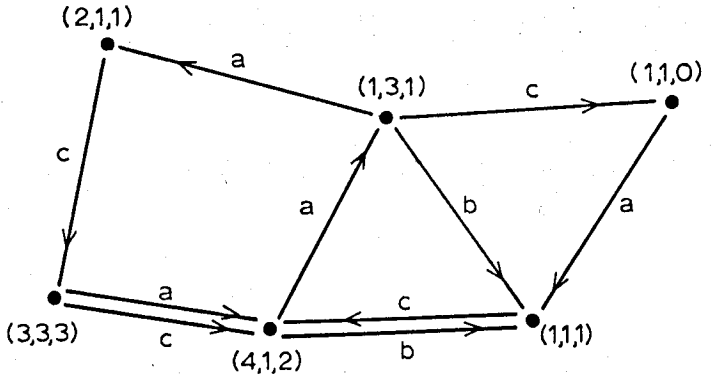


FIG. 1. An example of a problem graph. The nodes correspond to problem states, and the arcs to operators (transformations).

the node for which the sum of the triad of integers was least. In a path-search problem we seek a path between two states, that is, a sequence of operators which take us from one state to another. Referring again to Fig. 1, we might require a path from the node (3, 3, 3) to the node (1, 1, 0). A solution would be (*a*, *a*, *c*). Note that the sequence of states (3, 3, 3), (4, 1, 2), (1, 3, 1), (1, 1, 0) does not immediately give the corresponding operators (though the converse is true) and thus is a solution of a weaker kind. To appreciate this point the reader must realise that although in Fig. 1 he can see the whole of the problem graph laid out before him, the program must work from the definitions of the states and the operators, and the application of the latter to the former.

Before describing the Graph Traverser search method, it must be emphasised that these problems are always to be regarded from a practical rather than a theoretical standpoint. Actual solutions must be found to actual problems within a reasonable search time. A further point is that in a given problem situation the way in which the states and operators are defined is likely to be of great importance. Although this aspect of the matter is outside the present program schema, it must not be overlooked. In particular, for a state-search problem, the definition of the operator set may be an important part of the method of solving a problem, rather than of the problem situation itself.

THE GRAPH TRAVERSER SEARCH METHOD

The program proceeds by selecting an initial state either as directed or at random or, in a path-search problem, from the specification of the problem, and then by applying operators to this state and to the new states it thus generates, until the fragment of the problem graph which it has built up provides the information needed to find the solution. The structure which is constructed by the program is a 'tree', since although the program may generate a state more than once, it will only store it once, and will only remember one path to it.

When growing its search tree the program repeatedly encounters two problems. The first is to decide to which state next to apply operators. The second is, having chosen a state, to decide which operators actually to apply. The program's reaction to the first problem involves the 'evaluation function', and to the second the 'develop procedure'.

Briefly, the evaluation function must assign to each state encountered a value based on its structure. When a state must be selected for development (i.e., application of operators) that with the smallest value (smallest rather than largest since in path-search problems the value may often be identified with estimated distance from the 'goal') is chosen, avoiding states which have already been fully developed.

When a state has been chosen for development the develop procedure must decide two things, (i) which operators of those applicable actually to apply (this is the process of *operator selection*) and then having applied them (ii) whether or not to mark the state as fully developed. If a state is marked as fully developed then it will never be chosen for development again. If it is not so marked then the program leaves itself the option of taking up development of that state again. A state such as this which is neither quite undeveloped nor fully developed we call *partially developed.*

Perhaps the simplest thing to do when developing a state is to apply all possible operators and be done with it. The earliest work with the Graph Traverser used this approach, which does not involve partial development. At the other extreme, at most one operator may be applied in a development, the order of application being fixed but arbitrary, and the state being considered as partially developed until the last operator has been applied. This approach, which has an important application to be discussed later in the paper, is illustrated in Fig. 2, which shows the program searching the problem graph of Fig. 1. The arrows are reversed corresponding to the actual structure of the search tree. The task is to find the node the sum of whose integers is least. The program arbitrarily selects (4, 1, 2) as its starting point. As indicated above it applies just one operator during a development, and will not mark a node fully developed until all the operators which are applicable there have actually been applied. The order of application is always *a, b, c.* The value assigned to a node by the evaluation function is just the sum of its integers. The letters U, P, F indicate the status of a node—undeveloped, partially developed, or fully developed.

It is a consequence of this algorithm that the program carries out repeated developments of a node until either a "descendant" node of lower value is found, or no more applicable operators are available. In the latter event a "disconnected" development will ensue. Notice in (*f*) of Fig. 2 that when a

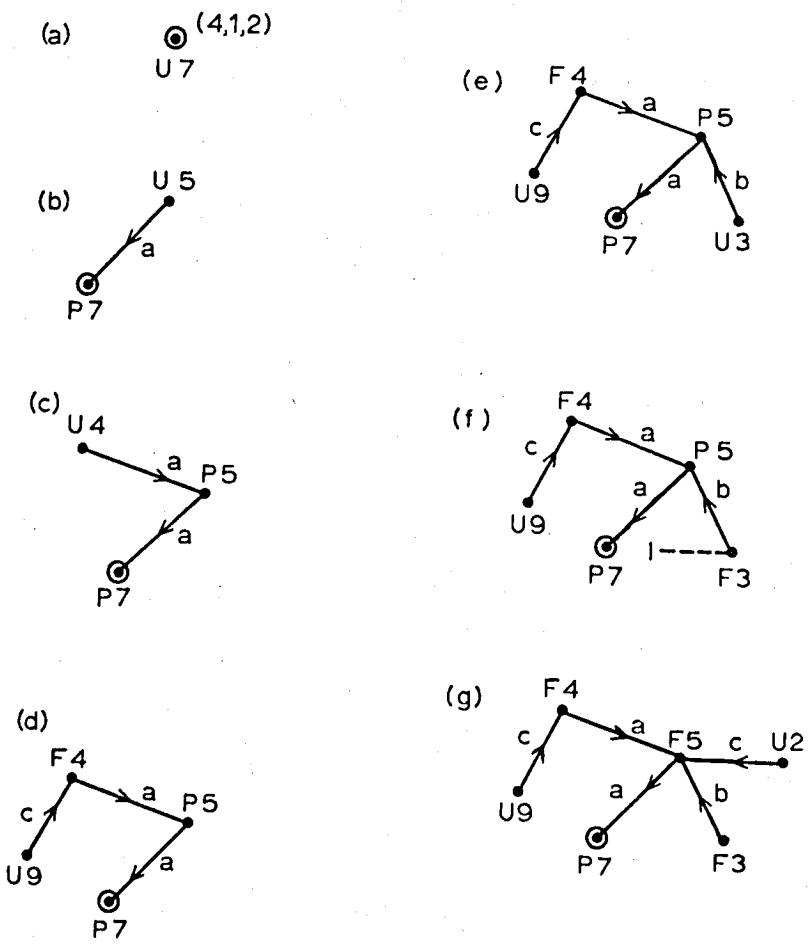


FIG. 2. The Graph Traverser searching the graph of Fig. 1. Successive diagrams show successive developments. The node from which the search starts is ringed for clarity. The letters U, P, F indicate the status of a node—undeveloped, partially developed, or fully developed. The figures give the values calculated for the nodes. The arcs are reversed since in the search tree the program stores a pointer to the ancestor of a node, not pointers to its descendants.

node already on the tree is generated its regeneration is ignored. Had the new encounter provided a shorter path to the node than that already known, the tree would have been modified accordingly. The search ultimately succeeds, but whether the best node would be found in practice depends upon the particular rule used to decide when to terminate the search.

This brings us to the most important point that these various decisions which the program must make depend heavily upon guidance from the user in a particular problem situation, although ways of making the program itself improve its decisions have been suggested elsewhere (Doran 1967, Michie 1967). In practice the develop and evaluate procedures, and any other routines which need to be respecified, are programmed for each application and embedded within the program proper.

THE TRAVELLING SALESMAN PROBLEM

The Travelling Salesman is one of the best known problems of operations research. It is very simply stated. A salesman, starting at a given city must visit $n-1$ other cities, each just once, before returning to base. There is a fixed distance to travel between any particular pair of cities and he wishes the whole tour to be as short as possible. The problem is to specify an efficient way for him to find the shortest tour. The word 'efficient' here is crucial. One can find the shortest tour by calculating the length of all possible tours. But if there are n cities, there are $\frac{(n-1)!}{2}$ possible tours, so that this approach is usually much too laborious.

This version of the problem corresponds to marking a score or so points on a piece of paper, and connecting them up so that the tour goes through each just once, and is as short as possible in length. More generally, we stop thinking in terms of cities or points on a plane surface, and simply say that a cost is assigned to travelling between each pair of 'cities' (independent of direction) in some way which does not concern us.

Linear programming and dynamic programming methods can be applied to the Travelling Salesman problem but only with difficulty, particularly if the number of cities involved is more than 20, say. Yet the problem is one of real importance, and in modified and disguised forms occurs repeatedly in industrial and economic situations. We may, for example, think of ships visiting ports, or less obviously, of an automatic press punching a series of holes in specified locations in pieces of sheet metal.

Shen Lin (1965) and G. M. Roe (1966) have recently explored with considerable success heuristic search methods which provide near-optimal tours, but avoid spending much time trying to turn a near-optimal tour into an optimal tour. By an 'optimal tour' I mean any tour with least possible cost.

Lin's approach, which is the one I shall discuss here (Roe's is broadly similar), defines for the Travelling Salesman problem a certain set of, in our terminology, operators. These are called (using Roe's terminology now) 'cuts'. The set of all n-cuts is the set of all possible ways of breaking any n links of a given tour and putting it together again, as a single loop, in some other way. In particular, a 2-cut involves reversing the direction in which a sequence of the cities is visited, and a 3-cut involves removing a sequence of cities from the tour and then reinserting this sequence, possibly reversed, at the same or some other point. If there are, say, 20 cities in a tour then there are 170 possible

2-cuts, and about 5000 possible 3-cuts. Lin uses 3-cuts (his experiments indicate that 3 is the best value of n for this purpose) to convert the set of all possible tours into a problem graph in the Graph Traverser sense, and then adopts an algorithm now to be described.

Lin's algorithm involves a number of 'searches'. In each search an initial tour is selected at random, and the 3-cuts are systematically checked until one which will generate a new tour with lower cost has been found. This is then applied. The process is repeated with the new tour until an improvement

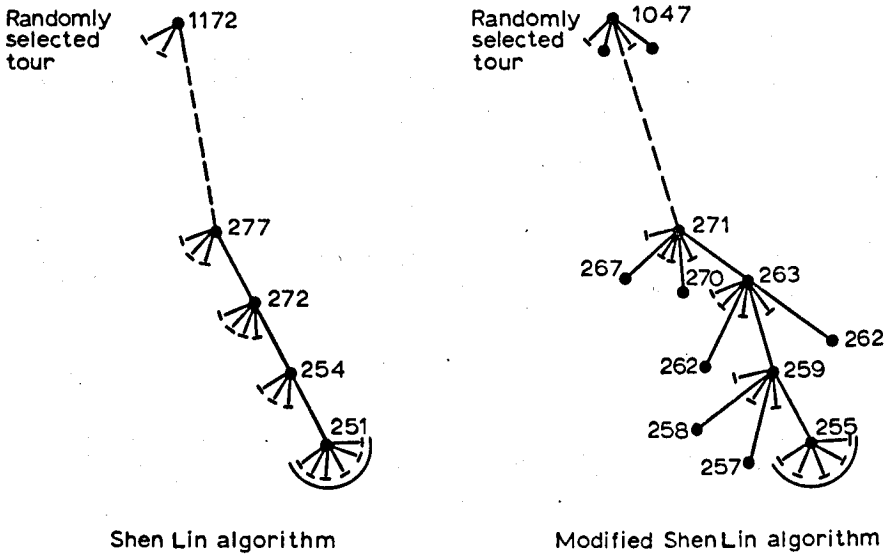


FIG. 3. Shen Lin's algorithm for solving the Travelling Salesman problem, and a modification of this to permit three new tours to be generated at each development. The nodes correspond to possible tours, and the figures give the corresponding costs. The blocked arcs indicate 3-cuts which do not reduce the cost. Both algorithms terminate when a tour is encountered which no 3-cut improves (a 3-opt tour).

is again obtained, and so on until ultimately a tour is reached which no 3-cut improves (Fig. 3). This tour is then the 'locally optimal tour' or '3-opt' tour associated with the initial random starting configuration. In all tens of searches will be made. Experiment shows that for all except the smallest problems, not all, and possibly none, of the 3-opt tours obtained will be optimal, nor in general will they be all distinct.

However, as Lin points out, after some experience with the algorithm it is possible to estimate in advance for an unsolved problem of particular size the probability that a search will give an optimal tour. It is then possible to calculate how many searches are necessary to reduce to a level low enough to be negligible the probability that an optimal tour still has not been discovered. In this situation it is not necessary to push on with an individual search until an optimal tour has definitely been located—a process likely to be both

inconclusive and very time consuming. Lin's program is for an IBM 7094 II, and is very fast, most relevant times being measured in seconds.

Lin also employs various techniques to use information collected in the earlier searches to cut computation in the later. These techniques will not concern us here but are interesting and important.

APPLICATION OF THE GRAPH TRAVERSER TO THE TRAVELLING SALESMAN PROBLEM

The Graph Traverser has been applied to the Travelling Salesman problem with two main objectives:

(i) to establish that it *is* applicable to this situation, and that all the basic results obtained by Lin can be obtained using the Graph Traverser, though in practice at a slower rate since the Graph Traverser aims at generality rather than speed; and

(ii) to use the flexibility and facilities of the Graph Traverser to explore the worth of other search methods more complex than that used by Lin.

The actual mapping process which takes Lin's algorithm into that of the Graph Traverser has already been indicated. A state of the problem graph corresponds to a possible complete tour (*not* to a city), and an operator to a 3-cut as already described. All the 3-cuts are applicable to any tour, but the develop procedure when called only actually applies one 3-cut—the first one which it considers which will, it calculates, generate a cheaper tour. The cuts are considered in an arbitrary but fixed order. To implement Lin's strategy, the develop procedure then marks the tour as fully developed. The effect of this is that when a 3-opt tour is encountered, so that no new undeveloped tour is generated before the current tour is declared fully developed, the search must terminate (Fig. 3).

Using the Lin algorithm which is an example of a 'conditional choice strategy' (see Michie 1967) the Graph Traverser was applied to randomly generated problems involving up to 20 cities, and to two published problems, the 20-city problem of Croes (1958), and the 25-city problem of Held and Karp (1962). The results obtained were consistent with those obtained by Lin. For the Croes problem Lin quotes 13 out of 40 for the proportion of successful searches, and for the Held and Karp problem 26 out of 40. The corresponding figures using the straightforward Graph Traverser implementation of Lin's algorithm were 3 out of 10 and 13 out of 16. A typical search on the 20-city problem involved developing about 45 tours, and on the 25-city problem about 70 tours. These results indicated that the first research objective had been achieved. An Elliot 4120 processor was used to obtain these and the other results described in this paper.

MODIFICATION OF SHEN LIN'S ALGORITHM

How can Lin's algorithm be improved? From the standpoint of the Graph Traverser program it is a rather simple strategy, and presents a number of opportunities for modification.

The first modification which suggested itself was to require the develop procedure to produce, where possible, more than one cheaper tour. Since the cost of a tour is being used as its value, the program will always select the cheapest descendant tour for the next development. This trades the extra computation time required for each development against, hopefully, a more successful search. In practice, three descendant tours were generated at each development (see Fig. 3). Table 1, which refers to the Croes problem, shows

TABLE 1

Costs of 3-opt tours obtained with the Graph Traverser for a particular Travelling Salesman problem. The three algorithms used are described in the text. In every case tours with the same cost are identical.

CROES 20-CITY PROBLEM
OPTIMAL TOUR HAS COST 246

Random starting tour	Lin algorithm	Modified Lin algorithm	Graph Traverser algorithm
A	246	246	246
B	251	246	251
C	252	252	246
D	252	252	246
E	252	255	246
F	246	252	246
G	246	252	246
H	252	246	246
I	252	252	246
J	251	252	246
Occurrences of optimal tour	3	3	9
Approximate time for each search in minutes	15-20	20	30

that each search took rather longer with no noticeable improvement in the results. This agrees with Lin's remark that 'Attention should be directed to finding improvements with a minimum amount of computation rather than to making the maximum improvement possible at each step'. What seems to be happening is that there are only a very few 3-opt tours, all quite similar, and which of these is encountered during any particular search depends only upon the final 'direction' of approach to the critical region. What happens earlier is quite irrelevant.

The mutual proximity of the 3-opt tours, in the sense that one can be converted into another by a very few 3-cuts (not, of course, by just one) suggests that terminating a search as soon as a 3-opt tour is encountered may be premature, and that a certain amount of additional investigation might be worthwhile—even though such additional investigation is likely to be very costly in time. Roe suggests something along these lines.

It seems appropriate to invoke the Graph Traverser's ability to partially develop tours, leaving them available for further development at a later time.

This process was illustrated in detail in Fig. 2. Specifically, instead of the develop procedure applying the first 3-cut it encounters which will reduce the tour cost, and then marking the tour as fully developed, this latter step is omitted unless there exists no unused beneficial operator at all. In consequence, when a 3-opt tour is found, the search does *not* terminate, but continues by restarting the development of the tour from which the 3-opt

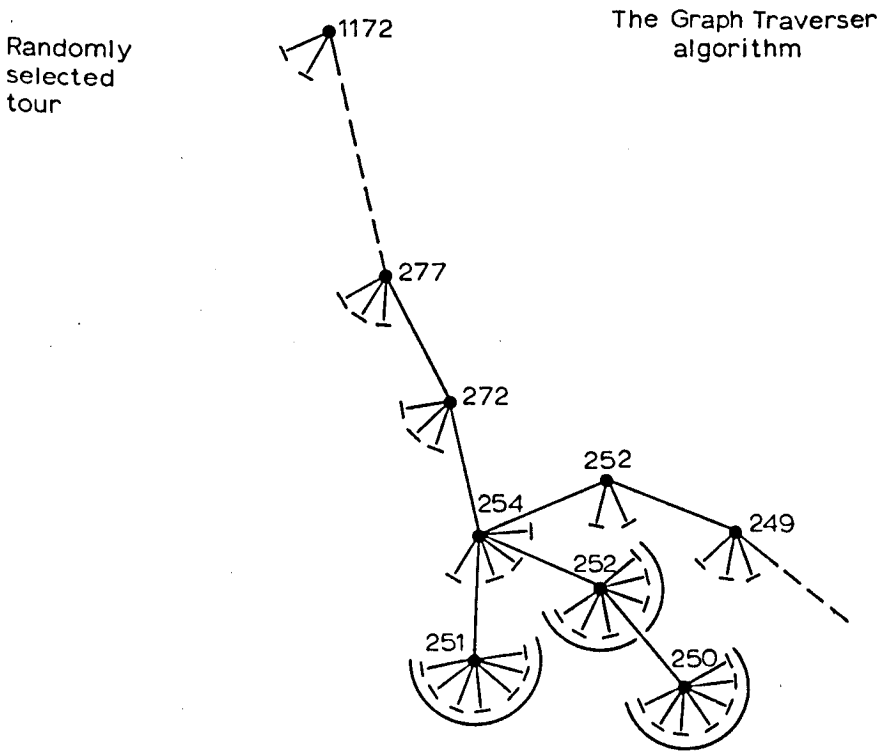


FIG. 4. Shen Lin's algorithm augmented by the use of the Graph Traverser's partial development facility. A search is continued beyond the discovery of a 3-opt tour by continuing the development of the best partially developed tour. The branches are explored from left to right, and the search is terminated when the program has been forced to 'retreat' a specified number of times.

tour was itself generated (Fig. 4). The search is continued in this way until some fixed number of tours has been fully developed. Since each full development corresponds to the program finding itself forced to 'retreat' to a tour of greater cost, this is a convenient and reasonable way to terminate a search. Of course, the 'result' of such a search is the cheapest tour encountered, not merely the last encountered.

EXPERIMENTAL RESULTS

Table 1 shows results obtained by this revised algorithm. For this problem, of which Lin remarks that it 'seems harder to solve than most 20 city prob-

lems', it almost uniformly succeeds in finding the best tour when the Lin algorithm fails. A search was terminated at the fourth retreat, which roughly doubled the time required for it.

Thus, although the average effectiveness of the searches has been increased, the time required for each has also risen. To decide whether there has been an overall improvement in performance we look at the matter in terms of the question 'How long must we run the program on this problem to achieve some selected probability that we have obtained an optimal tour?' Using the Lin algorithm the best estimate of the probability that a particular randomly initiated search will produce an optimal tour is clearly $p_1 = 0{\cdot}3$. It follows that the number of searches required to ensure a probability of 0·05 or less, say, that an optimal tour has not been found is found by solving the inequality in N_1

$$(0{\cdot}7)^{N_1} \leqslant 0{\cdot}05$$

This gives $N_1 \geqslant 9$. Supposing unit time for each search, the time required is 9 units.

Using the Graph Traverser strategy the estimated probability of a search succeeding is $p_2 = 0{\cdot}9$. The corresponding calculation gives $N_2 \geqslant 2$. Since the search time doubles, the time required is 4 units. Thus the time required is reduced by a factor of more than 2. The reader will easily see that this factor is broadly independent of the probability of error initially specified.

Thus the conclusion to be drawn from this admittedly very small quantity of evidence is that the partial development extension of Lin's algorithm is worthwhile.

Further investigation of this conclusion required results gathered from a range of problems. However this led to a snag. Fairly *difficult* problems were required, for otherwise Lin's basic strategy leaves little room for improvement. Suitably difficult problems were too large, however, for the limited machine store size immediately available.

An experiment was therefore carried out using 15 city problems defined by randomly generated cost matrices, and using *2-cuts* as operators rather than 3-cuts. The 2-cuts are a smaller, less drastic set of operators which Lin found to be less powerful in practice even allowing for the sharp reduction in the time required for each search when using them. Thus small problems which would otherwise be too easy are made suitably difficult by reducing the effectiveness of the operator set.

Eleyen random 15-city problems were generated, and ten searches were carried out on each, each search being continued until the 30th retreat occurred. Table 2 summarises the results obtained. The following points are important:

1. Column 2 gives the average number of cuts applied or considered for application by the program before reaching the nth retreat in the search, where n is indicated by the figure in column 1. Just over 300 operations are needed to reach the first retreat—the point at which Lin's search would

terminate—and thereafter just under 100 are on average needed to reach each successive retreat. These figures are effectively a measure of *time*.

2. Column 3 gives the mean cost of the best tour obtained up to the nth retreat and column 4 the observed probability that an optimum cost tour has already been found. What *was* the cheapest cost was decided by inspection of all the results obtained for the given problem. In no case was the matter in doubt. We see that the probability that a Lin search would be successful is about 0·09.

Note that there appears to be a sharp improvement if the search is carried only a little beyond the point where Lin terminates.

TABLE 2

Results obtained by applying the Graph Traverser algorithm to eleven randomly generated 15-city problems. The means are calculated over 110 searches in all. The final column refers to a 'typical' problem of this type and envisages several searches each terminating after a fixed number of retreats. The basic Lin algorithm would halt at the first retreat.

Retreat at which search ends	Mean number of operations per search	Mean cost of cheapest tour found in search	Observed probability that optimal-cost tour has been found in search	Total number of operations needed to give 0·99 probability of optimal tour
1	317	201	0·09	15316
2	416	193	0·22	7783
3	517	190	0·31	6439
4	605	187	0·34	6795
5	704	186	0·36	7172
6	798	185	0·36	8130
7	902	184	0·38	8636
8	995	183	0·43	8221
9	1090	182	0·47	7842
10	1176	181	0·49	8021
15	1651	179	0·55	9643
20	2105	178	0·60	10579
25	2570	177	0·65	11413
30	3025	177	0·68	12165

3. The last column attempts to combine columns 2 and 4 to discover if the loss in time is balanced by the gain in power. The figures give the number of operations, that is the number of inspections or applications of 2-cuts, needed to give a probability of 0·99 that an optimum tour has been located, assuming that we are faced with a new 15-city problem generated in the same way. The total number of operations required with termination on the rth retreat is $-2i_r/\log(1-p_r)$ where p_r is the estimated probability that an optimum tour will be found during a search thus terminated (column 4), and where i_r is the observed average number of inspections up to the rth retreat (column 2). These figures are artificial in two ways. First they assume that a non-integral number of searches is meaningful, in effect some smoothing has

been carried out, and secondly, and more important, it is assumed that all the problems generated by the random process are of equal difficulty. This latter assumption is certainly false. However, there is no reason to think that this will cause the figures to be misleading for our purposes.

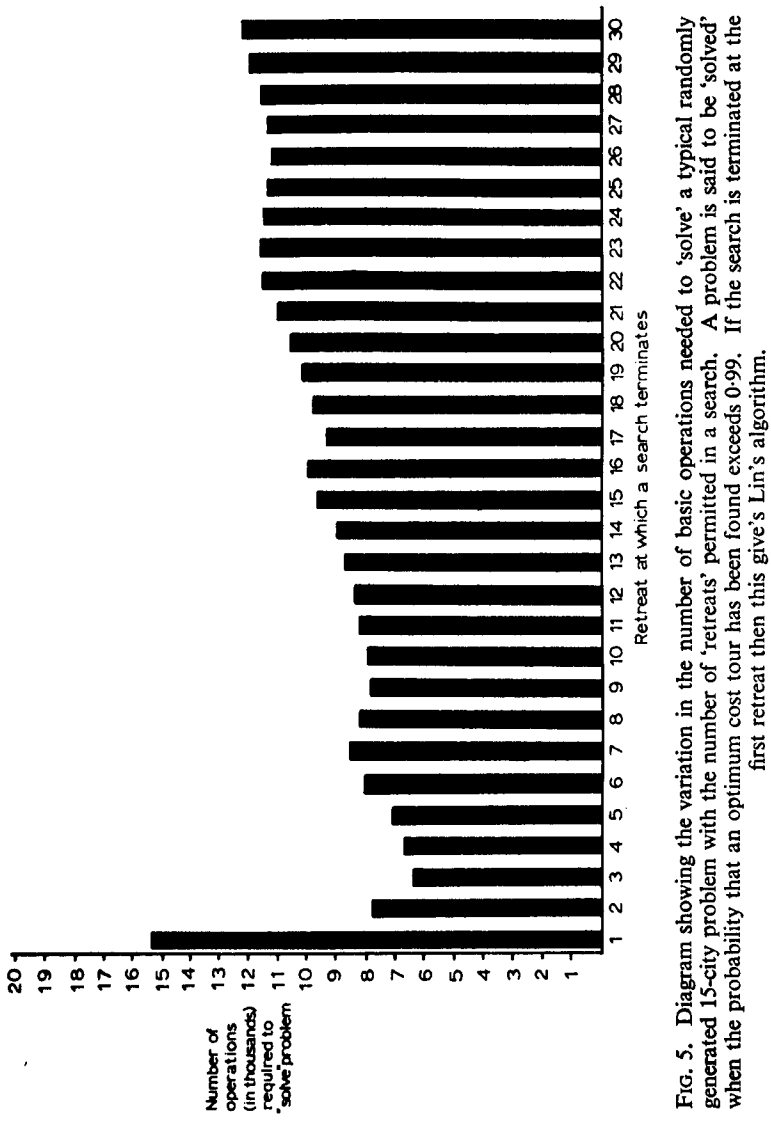


FIG. 5. Diagram showing the variation in the number of basic operations needed to 'solve' a typical randomly generated 15-city problem with the number of 'retreats' permitted in a search. A problem is said to be 'solved' when the probability that an optimum cost tour has been found exceeds 0·99. If the search is terminated at the first retreat then this give's Lin's algorithm.

Fig. 5 shows the required number of operations plotted against the length of search permitted. Again there is striking evidence in favour of continuing searches beyond the point at which Lin would terminate, and we see that the best results are obtained when a search is terminated at the third retreat. The apparently rhythmic fluctuation has no significance.

To sum up, although work is required on larger and more varied problems, it does appear that the Graph Traverser search method is capable of improving significantly Lin's already very successful algorithm. The improvement is obtained by continuing a search beyond the moment at which the first retreat occurs, that is, spending time searching the neighbourhood of locally optimal tours and proportionately reducing the total number of searches carried out.

Put another way, which may be more illuminating, the modification to Lin's algorithm is to start a proportion of the searches (as he defines them) from tours already known to be good, rather than always to start searches from randomly generated tours. This follows from the fact that when a retreat occurs, and development of an already partially developed tour is restarted, this is virtually equivalent to beginning a new search from that tour.

ON-LINE COLLABORATION WITH THE GRAPH TRAVERSER

It seems probable that in the foreseeable future problem-solving programs will be designed to interact with the user, rather than operate in isolation, when they are really intended to be useful. Such an arrangement permits the user to support the program in those aspects of the problem-solving procedure where it cannot cope adequately by itself. I shall therefore end this paper with a discussion of a version of the Graph Traverser adapted for such collaboration in the context of two simple sliding block puzzles, the Eight-puzzle and the Fifteen-puzzle, and with some more general remarks.

The Eight- and Fifteen-puzzles have been described in detail elsewhere (e.g., Doran & Michie 1966, Schofield 1967). Briefly, the Eight-puzzle consists of eight unit square pieces in a 3×3 frame, the pieces being numbered from 1 to 8, and there being one empty cell. The problem is to convert one configuration of the pieces into another by sliding the pieces around. Thus we might seek a sequence of *moves*, where a move means sliding one piece into the adjacent space, which converts

7	5	4		1	2	3
1	0	8	into	8	0	4
2	6	3		7	6	5

Provided that the two configurations regarded as permutations of the digits 0 to 8 have the same parity, at most 30 moves will be required.

The Fifteen-puzzle is the same puzzle scaled up to a 4×4 frame, and with the pieces numbered from 1 to 15. For this puzzle, although the same parity rule holds, the maximum number of moves needed to convert one configuration into another is not known.

Extensive results have already been published from the application of the Graph Traverser to these puzzles (Doran & Michie 1966, Doran 1967, Michie 1967). The problem is of the 'path-search' variety, unlike the Travelling Salesman application, and the evaluation function plays a much more important role, for it has the difficult task of estimating how close con-

figurations generated in the search are to the target configuration. As a result of this earlier work, fairly good evaluation functions are available for the puzzles.

A very simple way to bring a human solver into the scheme is to arrange that he takes over the function of the develop procedure, and this has been implemented. Every time the program wishes to develop a configuration the

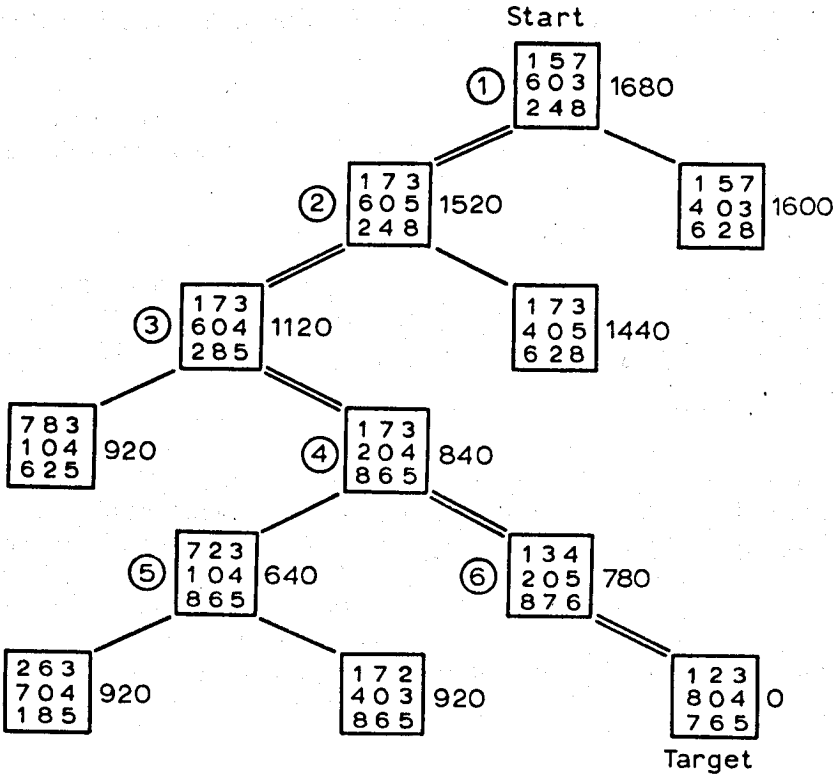


FIG. 6. The search tree constructed collaboratively by the Graph Traverser and the author to solve a particular Eight-puzzle configuration. The ringed figures give the order of development, and the unringed figures the values assigned by the evaluation function. Notice that after the fifth development the program rejects both the macro-moves suggested by the author in favour of a continuation previously itself rejected.

human solver is presented with it, and asked to specify sequences of moves (operators) which he suggests be applied. Once he has suggested all the moves or move sequences he thinks reasonable, he indicates this, the moves are implemented, and the configuration is henceforth regarded as fully developed. Thus there is no partial development.

From the user's point of view the sequence of events is as follows. The program asks whether it is the Eight-puzzle or the Fifteen-puzzle which is to be solved. Given the answer that it is the Eight-puzzle, say, the program then types out the standard goal configuration and asks the user to type the

starting configuration. This done, the program asks the solver what moves he suggests for this configuration. The solver might suggest three possible sequences of moves, and then type ALL. The program implements the moves, decides which of the three resulting configurations it favours, and presents this to the user for more moves to be suggested. This continues until either the goal is located, when the program will output the solution path with relevant statistics, or the solver types RESIGNS. The program will ignore impossible moves or typing errors.

The configuration selected by the program for 'human' development will not necessarily be the outcome of one of the moves or move sequences last suggested by the solver. As always with the Graph Traverser, disconnected developments are possible. That is, if none of the current suggestions of the solver is sufficiently attractive, the program will return to an earlier possible continuation not thought the best at the time. This point is illustrated in Fig. 6, which shows the search tree built up by the author and the 'on-line' Graph Traverser to solve a particular Eight-puzzle configuration. The nodes correspond to configurations and are labelled with the value assigned to them by the program, and with an integer indicating the order in which the developments occurred. The final solution path involved 26 basic moves, compared with a minimum possible of 22. In a parallel attempt on this configuration *not* making use of the program the author required 36 moves to solve it. Indeed, although no serious experimentation has been attempted, the author usually seems to benefit from the program's assistance, although whether the converse is true is less clear!

DISCUSSION

This example of a collaborative use of the Graph Traverser is of no great importance in itself, but it does serve to introduce some further ideas. Dr Donald Michie has pointed out to me that Travelling Salesman problems could be solved by an analogous collaborative approach. The user would be presented with a tour to which he would suggest modifications, and the program would next present either a tour generated by one of the suggested modifications, or a tour generated earlier in the search, but not then judged sufficiently attractive to be followed up.

More generally, a version of the program might be prepared in which the function of both the develop procedure *and* the evaluate procedure were taken over by an on-line user. Since all other aspects of the program's behaviour can be controlled by setting parameters, which even in the current version of the program can be done on-line, such a program would be capable of being applied, *without additional programming*, to a wide range of problems certainly including both of the puzzles mentioned and the Travelling Salesman. However, although this would certainly achieve generality, so much of the work load would have been transferred to the user and progress would be so slow, that the arrangement is unattractive.

What facilities would an on-line user of the Graph Traverser ask for?

Certainly he will not wish to do any preparatory programming. He will wish to enter the program, to specify to it the task, to give it such guidance as he can, and then either to withdraw entirely or to remain on hand to answer questions which the program may ask. He will often wish to monitor the program's progress and to change the problem specification, or to give additional guidance where appropriate. In Graph Traverser terms this means that the user must at least be able initially to specify the operator set, that is the develop procedure, and the evaluation function, and then to change his specification at will. Some kind of on-line problem-oriented programming language, however primitive, seems essential. This could be provided within Algol, but neither easily nor efficiently. It seems obvious that such a version of the Graph Traverser should be wholly written in a language designed for on-line usage. POP-1, created by R. J. Popplestone and described elsewhere in this volume, is attractive given the ease with which it permits the construction of a library of sub-routines.

Other papers already mentioned have suggested ways in which the Graph Traverser might itself improve its search apparatus, usually by some form of parameter optimisation, and thus be a more 'intelligent' problem-solver. D. Vigor, in his stimulating paper included in this volume, emphasises another aspect of intelligence by stressing the importance not of the way in which a program *attacks* a problem (or general game) situation, but of the way in which the program *represents* the situation. He suggests that very often this is the most important step in the solution process, and outlines a method for seeking the most efficient problem representation. I mention this because it indicates an aspect of problem-solving which, although unrecognised by the present Graph Traverser, must surely be important in future work.

ACKNOWLEDGEMENTS

The research described in this paper is sponsored by the Science Research Council. I am greatly indebted to Dr Donald Michie for many discussions on the topics covered by this paper and, in particular, for his original suggestion that the Graph Traverser might be applicable to the Travelling Salesman in the manner described.

REFERENCES

Croes, G. A. (1958). A method for solving Traveling Salesman problems. *Operations Research*, **6**, 791-812.

Doran, J. E. (1967). An approach to automatic problem-solving. *Machine Intelligence* 1, pp. 105-123. Collins, N. L., and Michie, D. (eds.). Edinburgh: Oliver & Boyd.

Doran, J. E., & Michie, D. (1966). Experiments with the Graph Traverser program. *Proc. R. Soc.* (A) **294**, 235-259.

Held, M., & Karp, R. M. (1962). A dynamic programming approach to sequencing problems. *J. Soc. Ind. Appl. Math.* **10**, No. 1, 196-210.

Lin, S. (1965). Computer solutions of the Travelling Salesman problem. *Bell System Tech. J.*, **44**, 2245-2269.

Michie, D. (1967). Strategy-building with the Graph Traverser. *Machine Intelligence* 1, pp. 135-152. Collins, N. L., and Michie, D. (eds.). Edinburgh: Oliver and Boyd.

Newell, A., & Ernst, G. (1965). The search for generality. In *Information Processing 1965: Proceedings of IFIP Congress 1965*. Vol. 1, pp. 17-24. Wayne A. Kalenich (ed.). London: Macmillan.

Roe, G. M. (1966). Three fast, suboptimal procedures for the Traveling Salesman problem. *General Electric* report No. 66-C-051.

Schofield, P. (1967). Complete solution of the Eight-puzzle. *Machine Intelligence* 1, pp. 125-133. Collins, N. L., and Michie, D. (eds.). Edinburgh: Oliver and Boyd.

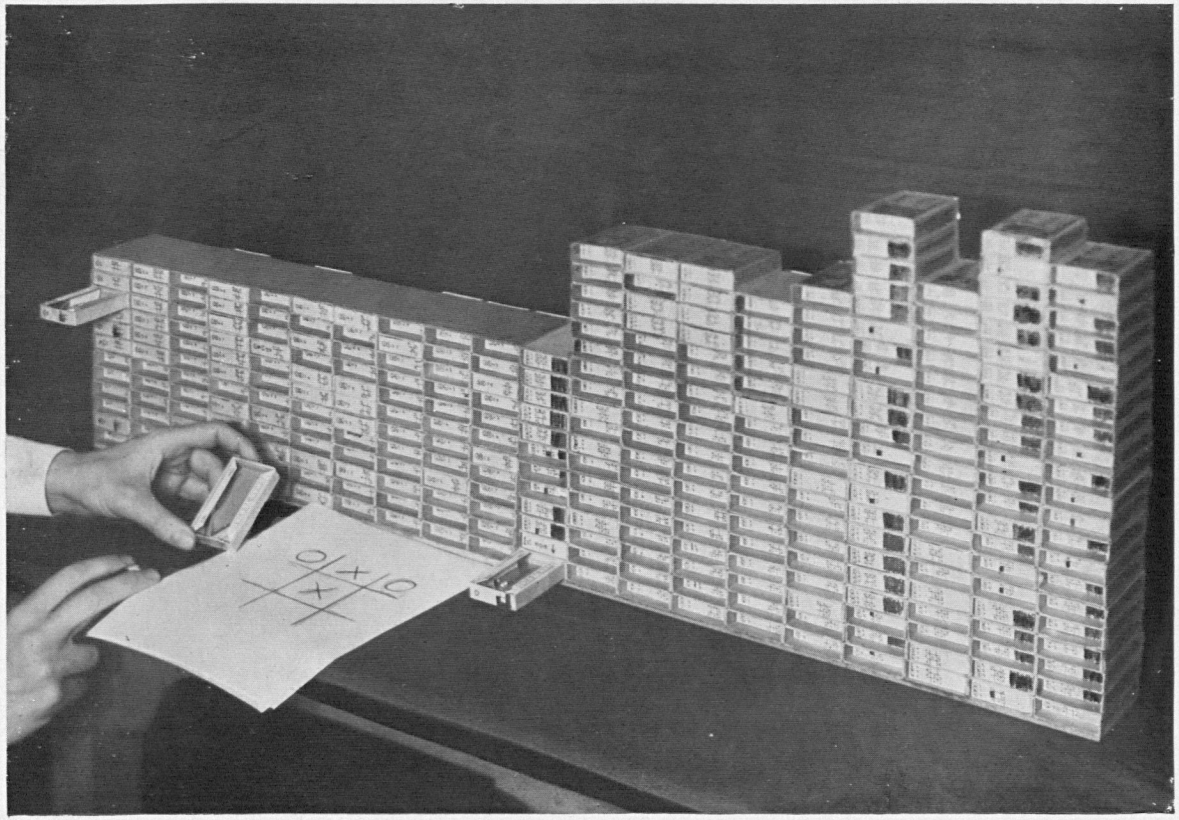


FIG. 1. The original matchbox version of MENACE

9

BOXES: AN EXPERIMENT IN ADAPTIVE CONTROL

D. MICHIE and R. A. CHAMBERS
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

BOXES is the name of a computer program. We shall also use the word boxes to refer to a particular approach to decision-taking under uncertainty which has been used as the basis of a number of computer programs.

Fig. 1 shows a photograph of an assemblage of actual boxes—matchboxes to be exact. Although the construction of this Matchbox Educable Noughts and Crosses Engine (Michie 1961, 1963) was undertaken as a 'fun project', there was present a more serious intention to demonstrate the principle that it may be easier to learn to play many easy games than one difficult one. Consequently it may be advantageous to decompose a game into *a number of mutually independent sub-games* even if much relevant information is put out of reach in the process. The principle is related to the method of subgoals in problem-solving (see Newell *et al.* 1960) but differs in one fundamental respect: subgoals are linked in series, while sub-games are played in parallel, in a sense which will become apparent.

DECOMPOSITION INTO SUB-GAMES

The motivation for developing algorithms for small games (by a 'small' game we mean one with so few board positions that a boxes approach is feasible) needs explanation, since small games are generally too trivial to be of intellectual interest in themselves. The task of *learning* a small game by pure trial and error is, on the other hand, not trivial, and we propose that a good policy for doing this can be made useful as a component of a machine strategy for a large game. The point is that the board states of a large game may be mapped

on to those of a small game, in a many-to-one mapping, by incomplete specification. This is what the chess player does when he lumps together large numbers of positions as being 'similar' to each other, by neglecting the strategically irrelevant features in which they differ. The resultant small game can be said to be a 'model' of the large game. He may then, in effect, use his past experience *of the model* to select broad lines of play, and in this way guide and supplement his detailed analysis of variations in the large game. To give a brutally extreme example, consider a specification of chess positions so incomplete as to map from the viewpoint of White the approximately 10^{50} positions of the large game on to the seven shown in Fig. 2. Even this simple classification may have a role in the learning of chess. A player comes to

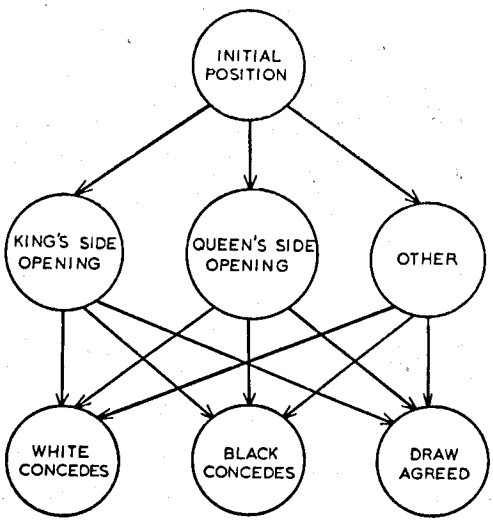


FIG. 2. A 'model' of chess.

realise from the fruits of experience that for him at least it pays better to choose King's side (or Queen's side, or other, as the case may be) openings. The pattern of master play, it may be added, has also changed, over the past hundred years, in ways which would show up even in this model, for example through decrease in the frequency of King's side openings and increase in the frequency of draws. Before leaving the diagram we shall anticipate a later topic by observing that it has the structure of a three-armed bandit with three-valued pay-offs.

In brief, we believe that programs for learning large games will need to have at their disposal good rules for learning small games. In the later part of this paper we apply the idea to an automatic control problem viewed as a 'large game'. As will appear, the approach is crude but the results to be reported show that decomposition into sub-games, each handled in mutual isolation by the same simple decision rule, is in itself sufficient to give useful performance in a difficult task.

As an example of the 'boxes' approach, consider again the matchbox machine of Fig. 1. This consists of 288 boxes, embodying a decomposition of the game of tic-tac-toe (noughts and crosses) into 288 sub-games, this being the number of essentially distinct board positions with which the opening player may at one time or another be confronted. Each separate box functions as a separate learning machine: it is only brought into play when the corresponding board position arises, and its sole task is to arrive at a good choice of move for that specific position. This was implemented in MENACE by placing coloured beads in each box, the colour coding for moves to corresponding squares of the board. Selection of a move was made by random choice of a bead from the appropriate box. After each play of the game the value of the outcome ('win', 'draw' or 'lose') was fed back in the form of 'reinforcements', i.e., increase, or decrease, in the probability of repetition in the future of the moves which in the past had led to the good, or bad, outcome. Echoing the terminology of Oliver Selfridge (1959), whose earlier 'Pandemonium' machine has something of the 'boxes' concept about it, one may say that the decision demon inhabiting a given box must learn to act for the best in a changing environment. His environment in fact consists of the states *of other boxes.* He cannot, however, observe these directly, but only the outcomes of plays of the game which have involved his box.

How should each demon behave?

PROBABILISTIC DECISION BOXES

In MENACE, and its computer simulation, choices among alternative moves were made initially at random, and after each play the probabilities of those moves which the machine had made were modified by a reinforcement function which increments move-probabilities following a win, leaves them unchanged following a draw and diminishes them following a defeat. In an alternative mode the value of the outcome is measured not from a standard outcome as baseline but relative to the past average outcome. With this sliding origin feature, a draw is reckoned a good result when defeat has been the rule, but a bad result when the machine is in winning vein. Before leaving the topic we shall make two remarks about adaptive devices based on the reinforcement of move-probabilities:

(i) such devices cannot be optimal;
(ii) it may nevertheless be possible in some practical contexts, to improve the best deterministic devices that we know how to design by incorporating a random variable in the decision function.

The basis of the second remark is connected with the fact that a move has an information-collecting role, and that a trade-off relation exists between expected gain of information and expected immediate payoff. The existence of this relation becomes apparent as soon as we try to devise optimal deterministic decision rules to replace *ad hoc* reinforcement of move-probabilities.

DETERMINISTIC DECISION BOXES

The task then is to draw up optimal specifications for the demons in the boxes.

In the present state of knowledge we cannot do this, as can be seen by considering the simplest learning task with which any demon could possibly be faced. Suppose that the demon's board position is such that his choice is between only two alternative moves, say move 1 and move 2. Suppose that his opponent's behaviour is such that move 1 leads to an immediate win in a proportion p_1 of the occasions on which it is used and move 2 wins in a proportion p_2 of plays. p_1 and p_2 are unknown parameters, which for simplicity we shall assume are constant over time (i.e., opponent does not change his strategy). The demon's task is to make his choices in successive plays in such a way as to maximise his expected number of wins over some specified period.

Under these ultra-simplified conditions, the problem is equivalent to the 'two-armed bandit' problem, a famous unsolved problem of mathematics. The difficulty can be expressed informally by saying that it can pay to make a move which is, on the evidence, inferior, in order to collect *more evidence* as to whether it really is inferior. Hence the problem can be formulated as that of costing information in the currency of immediate gain or loss: how much is the choice of a given move to be determined by its cash value for the current play of the game (we assume that all games are played for money) and how much by its evidence-collecting value, which may be convertible into future cash? We propose now to illustrate this formulation by exhibiting the behaviour of a game learning automaton designed by D. Michie to be optimal in all respects except that the evidence-collecting role of moves is ignored. R. A. Chambers' program which simulates the automaton is known as GLEE (Game Learning Expectimaxing Engine).

The learning automaton starts with a full knowledge of the moves allowed it and the nature of the terminal states of the game, but it is initially ignorant of the moves available to the opponent and only discovers them as they are encountered in play. As they are encountered, new moves are recorded and thereafter frequency counts are kept to record their use by the opponent.

Each terminal node of the game-tree represents a result of the game; in the application of GLEE to the game of Noughts and Crosses, a terminal node is given a utility value of $+1$ if the result is a win for the automaton, -1 for a loss and 0 for a draw. Non-terminal nodes are assigned scores which estimate the expected outcome value of the game. The automaton assigns these scores in the light of its previous experience and revises them as its experience is increased. Initially the scores are zero for all non-terminal nodes.

The graph is considered by levels, all nodes on the same level being produced by the same number of moves. Thus nodes at level one represent the states of the game produced by the opening move. Positions which can be equated by symmetry are represented by the same node (as is also the case with the MENACE automaton).

The revision of scores, i.e., the update, is done by backward analysis starting from the terminal node where the last game ended. The process moves back through the graph one level at a time. For levels where the automaton is on play each node is given a score equal to the maximum of the scores of nodes that can be reached by one legal move. Where the opponent is on play, his previous behaviour from the given node is considered and the node is assigned a score equal to the *expected value* of the outcome. We call this updating process 'expectimaxing'. The expected value calculation is derived as follows:

Consider a node, N, at which the opponent is to move and let him have used k different moves, each n_i $(i=1, \ldots, k)$ times in the past. The total number of alternative plays from N is unknown so we assume that c additional moves exist and reach nodes of score zero (the value 2 was taken for c in the experimental runs referred to later). The other moves end in nodes with scores s_i $(i=1, \ldots, k)$.

By a development of Laplace's Law of Succession we can determine the probability, p_i, that the next opponent move from N will use the ith alternative, i.e.,

$$p_i = \frac{n_i+1}{\text{Estimated number of alternatives} + \text{total number of past plays from } N.}$$

$$= \frac{n_i+1}{(k+c)+\sum_{i=1}^{k} n_i}$$

Knowing those values of p_i $(i=1, \ldots, k)$ and the scores reached by each alternative we can calculate the quantity.

$E=\sum_{i=1}^{k} p_i s_i$. This defines the score associated with the node N.

To make a move the automaton examines all the legal alternatives and chooses the move leading to the position having the highest associated score, ties being decided by a random choice. It thus seeks to optimise the expected outcome of the *current* play of the game only. Fig. 3 shows the results of three trials each of 1000 games in which the opening player was a random move generator and the second player was the learning automaton. The automaton's score is shown as the number of wins minus the number of losses over each 100 consecutive games. By analysis of the optimal move trees associated with each type of opening play, the score level for an optimal strategy was calculated. It can be seen from Fig. 3 that the performance of the learning automaton levels out below this optimal level. This is due to an inherent weakness in the automaton's aim of short-term optimisation. The weakness is that as soon as the automaton finds, in a given position, a move with a positive expected outcome, then since other tried moves have had negative outcomes and unknown moves have zero expected outcomes it will continue to use that move as long as its expected outcome stays just greater than

zero. In this way it may be missing a better move through its lack of 'research'. Thus we see that neglect of evidence-collecting can lead to premature decision taking. This can be demonstrated in detail by reference to the very simple game of two-by-two Nim.

In this trivial case of the game there are two heaps of two objects and each of the two players in turn removes any number of objects from any one heap. The winner, according to the version considered here, is the player to remove

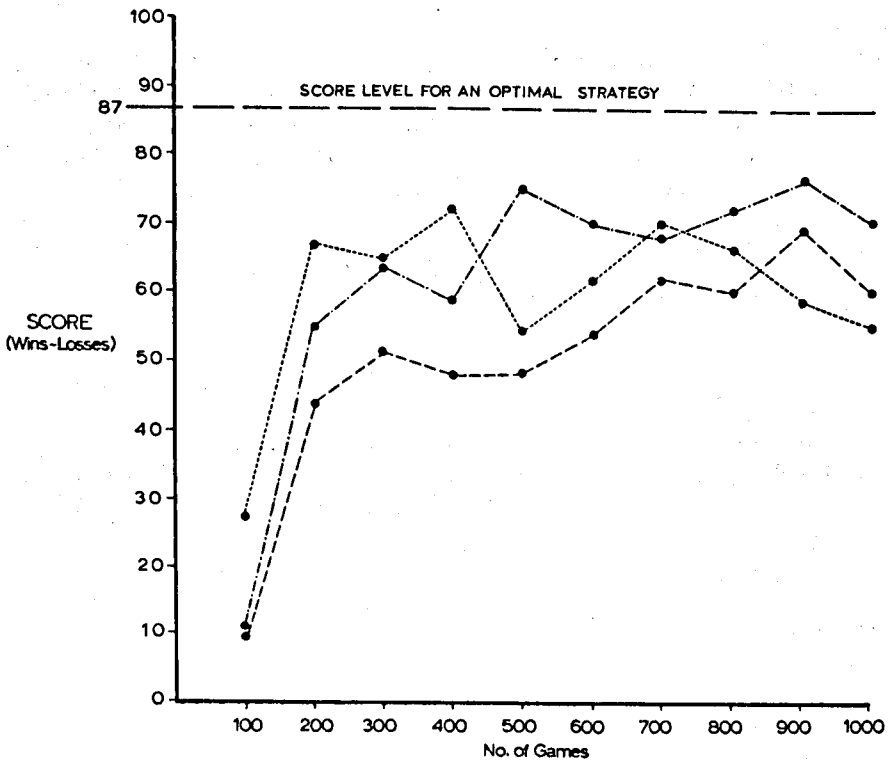


FIG. 3. Some representative results of GLEE playing Noughts and Crosses as second player, the opening player being a random move generator. The vertical scale gives hundred game totals.

the last object. Let the automaton be the opening player, then symmetry reduces the choice of opening moves to two. Against a good opponent the automaton will always lose the game, and both openings will be equally bad. But consider the case of a random opponent. Fig. 4 shows a graph representation of the game. For his first reply the opponent has three alternative moves from node (A) and two from node (B). As the opponent plays randomly, games proceeding from (A) will on average give the automaton two wins for every loss while those games proceeding from (B) will yield equal numbers of wins and losses. Thus in the long term the automaton's best opening move is to select position (A). Consider now a sequence of four games as follows:

the automaton chooses (A) and loses, (B) and wins, then (B) twice more losing each time. Fig. 5 shows the state of the automaton's knowledge with the assumed and as yet unknown opponent moves. Applying expectimax, as

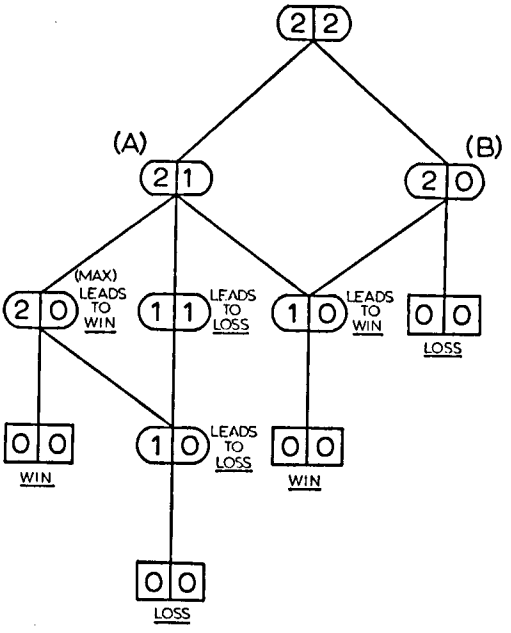


FIG. 4. 2 × 2 NIM showing winning and losing moves. The integer pairs at nodes indicate the state of play by showing the number of items in each of the two heaps.

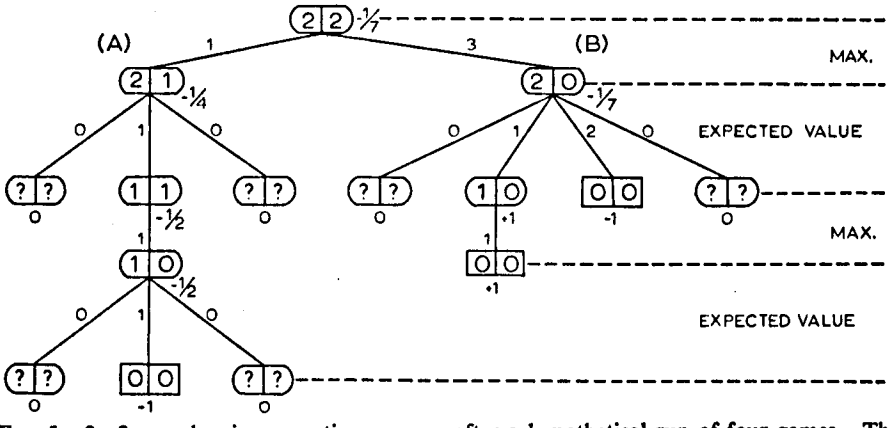


FIG. 5. 2 × 2 NIM showing expectimax scores after a hypothetical run of four games. The numbers against nodes are scores and the integers against graph connections show move frequencies.

explained, shows that due to the premature decision taking, brought on by losing at (A) and winning first time at (B), the automaton will still choose (B) despite the last two losses.

This feature of premature decision taking was also very evident in the trials of the automaton at noughts and crosses against a random opponent. Table 1 illustrates a series of games in which the automaton was second player and shows all the games in the first 100 of the series in which the random opponent opened in the centre. Analysis of this opening shows that by symmetry there

TABLE 1

Some selected results from the GLEE game-learning program using the game of Noughts and Crosses (tic-tac-toe). The automaton is the second player against a random opponent. Only those plays are shown in which the opponent opened to the centre of the board. For further explanation see text.

No. of each game in which random opponent opened in centre	Automaton's first reply: M=middle of side C=corner	Result for automaton
7	C	Lost
12	C	Lost
23	C	Lost
26	M	Won
27	M	Won
28	M	Won
31	M	Drew
37	M	Won
39	M	Lost
40	M	Won
44	M	Won
55	M	Lost
56	M	Won
61	M	Won
62	M	Lost
65	M	Lost
66	M	Lost
67	M	Lost
69	M	Won
74	M	Lost
77	M	Won
80	M	Lost
83	M	Lost
87	M	Won
89	M	Drew
90	M	Lost
92	M	Lost
95	M	Won
98	M	Drew
100	M	Lost

are only two distinct replies, a corner, C, or the middle of a side, M, the first being superior to the second. Indeed against an optimal strategy the second player always loses after playing M. However against a random opponent the automaton chanced to lose three games using the first reply, C, and then to win three using the second, M. This was sufficient to cause it to play the second, poorer, move throughout a trial of 2000 games. In the first 1000 games

the random player opened 336 games with the centre move with the automaton replying by moving into the middle of a side, M. In a similar series of 1000 there were 320 games with centre opening but this time the automaton always played in the corner, C. The results as percentages were as follows:

Automaton replying M 62% wins 18% losses 20% draws
Automaton replying C 75% wins 16% losses 9% draws

A remedy for this fault, which illustrates again the information-versus-payoff dilemma of trial-and-error learning, could be developed by use of a method which we have incorporated in our application of the boxes idea to a problem in adaptive control. The program is called BOXES and the method is called the 'target' method.

ADAPTIVE CONTROL AS A STATISTICAL GAME

BOXES is based on formulating the adaptive control problem in terms of the 'game against nature'. A valuable and detailed survey of formulations of this type has recently been published by Sworder (1966), who restricts himself in the main to theoretical questions of optimality. The boxes algorithm was devised by D. Michie for tasks for which optimal policies cannot be specified, and implemented in 1961 by Dean Wooldridge, junior, as a FORTRAN II program. It is incomplete in a number of ways. Yet it illustrates clearly the idea, expressed above, of reducing a large game to a small, model, game and then handling each separate board state of the model as a separate sub-game (box). In the adaptive control situation, where the state variables are real numbers, the large game is infinitely large, so that the sacrifice of information entailed in the boxes approach is correspondingly extreme. In spite of this, the algorithm actually does master the simulated task, which is one which severely taxes conventional methods.

BALANCING A POLE

The model task chosen for experimentation was that of balancing a pole on end. For the purpose in hand, this system has a number of merits:

(1) it has been thoroughly studied by Donaldson (1960) in an illuminating exercise on a related, but different, theme—namely the design of an automaton to learn a task by sensing the control movements made by a second automaton already able to perform this task (e.g., a human being);
(2) Widrow and Smith's (1964) subsequent and independent approach to Donaldson's problem used a design reminiscent of the boxes principle;
(3) a recent study in automatic control (Schaefer and Cannon 1966) has shown that the pole-balancer problem generalises to an infinite sequence of problems of graded difficulty, with 1, 2, 3, . . . , etc., poles balanced each on top of the next. There is thus ample scope, when needed, for complicating the problem;

(4) even the 1-pole problem is extremely difficult in the form for which adaptive methods are appropriate, i.e., physical parameters of the system specified in an incomplete and approximate way, and subject to drift with the passage of time. No optimal policy is known for such problems in general.

Fig. 6 shows the task. A rigid pole is mounted on a motor-driven cart. The cart runs on a straight track of fixed length, and the pole is mounted in such a way that it is only free to fall in the vertical plane bounded by the track.

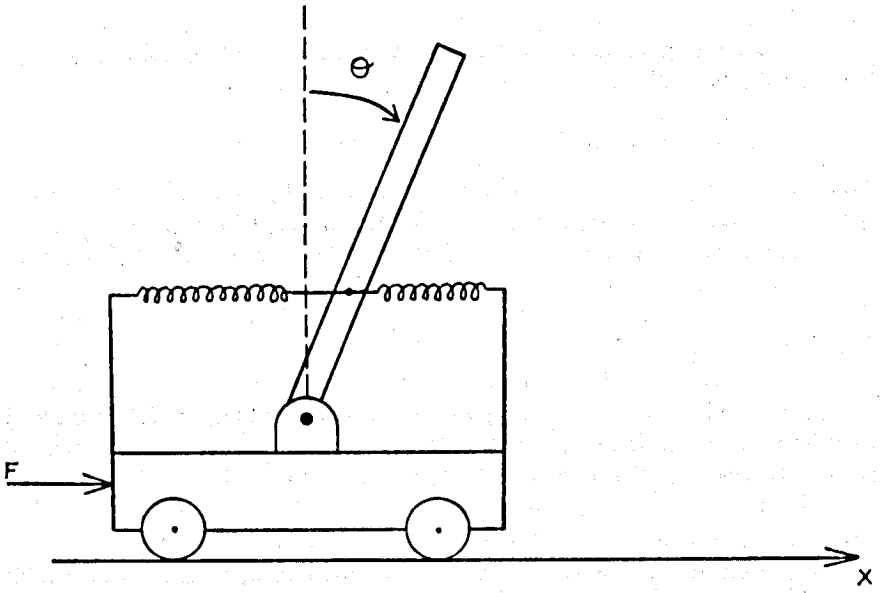


FIG. 6. A simple representation of the system being controlled. The spring is used to reduce the effective force of gravity. This feature was incorporated in the original laboratory apparatus so as to bring it within the range of human powers of adaptive control.

The cart's motor applies a force which is constant except for the sign. The sign is controlled by a switch with only two settings, + and −, or 'left' and 'right'. The problem is thus one of 'bang-bang' control. For the experimental runs reported here the cart and pole system was simulated by a separate part of the program, not built in hardware. The interval from sense to control action was set to zero and the interval from control action to sense to 0·05 sec. Various other parameters of the simulation program corresponding to such quantities as frictional resistance, length of pole, the mass of the pole and of the cart, and the spring clamping of the pole, were adjusted until the behaviour of the simulated system approximated to that of a real physical system.

BLACK BOX

The form in which we pose the problem is this. The adaptive controller must operate without prior knowledge of the system to be controlled. All it knows

is that the system will emit signals at regular time intervals, each signal being *either* a vector describing the system's state at the given instant *or* a failure signal, indicating that the system has gone out of control. After a failure signal has been received, the system is set up afresh and a new attempt is made. On the basis of the stream of signals, and this alone, the controller must construct its own control strategy. No prior assumptions can safely be made about the nature of the system which emits the signals, nor of the time-invariance of its parameters. In this sense, the task is never finished, since parameters of the system might drift into a new configuration requiring further modification of the controller. Most 'black box' problems permit some time-invariance assumptions. Our program must be ready for an even blacker box.

STATE SIGNALS

The state of the system at any instant can be represented by a point in an n-dimensional space of which the axes correspond to the state variables, n in number. In our case it is convenient to recognise four such variables,

(i) x, the position of the cart on the track,
(ii) θ ,the angle of the pole with the vertical,
(iii) $\dot{x}$, the velocity of the cart, and
(iv) $\dot{\theta}$, the rate of change of the angle.

(iii) and (iv) could be estimated by differencing (i) and (ii) with respect to time interval, but the program has these sensed separately. The state signal thus consists of a 4-element vector.

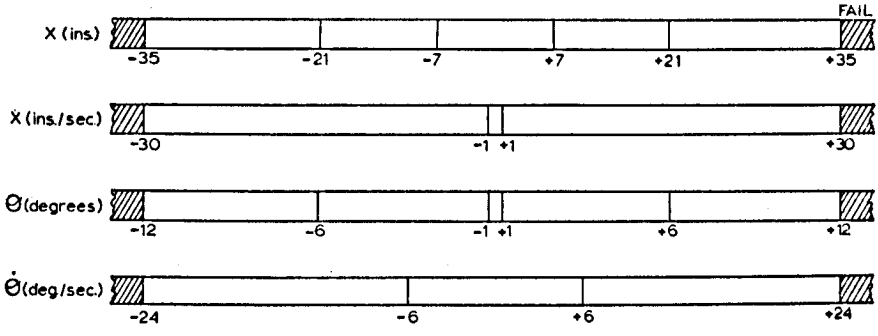


FIG. 7. Thresholds used in quantizing the state variables.

The 'model' was constructed by quantising the state variables by setting thresholds on the four measurement scales as shown in Fig. 7: thus, only 5 grades of position, x, were distinguished, 5 of angle, 3 of velocity and 3 of angel-change. It can be seen that this quantisation cuts the total space into a small number of separate compartments, or boxes; in our implementation the number was $5 \times 5 \times 3 \times 3 = 225$.

DECISION RULES

In order to envisage how the rest of the algorithm works it is easiest to imagine each one of the 225 boxes as being occupied by a local demon, with a global demon acting as a supervisor over all the local demons, according to the

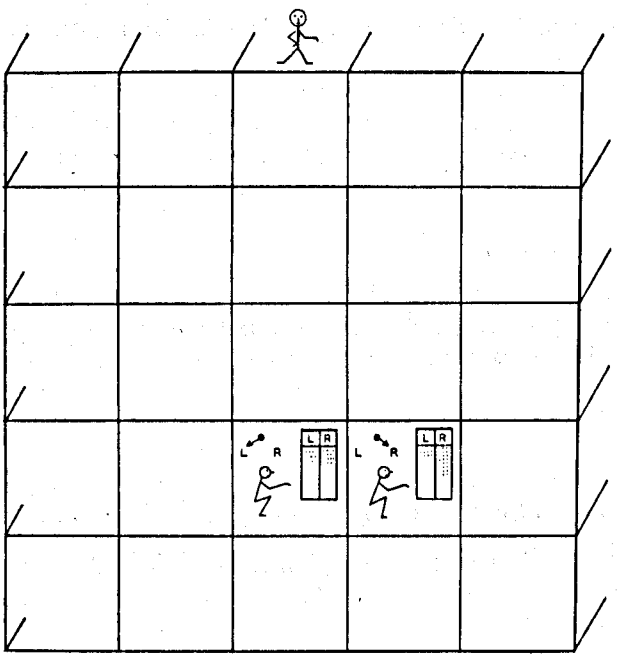


FIG. 8. A simple representation of the independent decision mechanisms controlling each box in the state space.

general scheme of Fig. 8. Each local demon is armed with a left-right switch and a scoreboard. His only job is to set his switch from time to time in the light of data which he accumulates on his scoreboard. His only experience of the world consists of:

(i) *LL* the 'left life' of his box, being a weighted sum of the 'lives' of left decisions taken on entry to his box during previous runs. (The 'life' of a decision is the number of further decisions taken before the run fails.)

RL the 'right life' of his box.

LU the 'left usage' of his box, a weighted sum of the number of left decisions taken on entry to his box during previous runs.

RU the 'right usage' of his box.

(ii) *target* a figure supplied to every box by the supervising demon, to indicate a desired level of attainment, for example, a constant multiple of the current mean life of the system.

(iii) $T_1, T_2, \ldots, T_i, \ldots, T_N$ times at which his box has been entered during the current run. Time is measured by the number of decisions taken in the interval being measured, in this case between the start of the run and a given entry to the box.

The demon's decision rule gives S, the setting of his switch, as a function of (i) and (ii). We can express this informally by saying that when the system state enters his box for the first time, the demon sets S by applying his decision rule, and notes the time of entry, T_1; if his box is re-entered during the run, the demon notes the new entry time, T_i, but does not change his switch, so that the same decision, S, is taken at every entry to his box during the course of that run. When the run fails, the global demon sends a failure message to all the local demons, indicating the time of failure, T_F, and also giving a new value of *target*. Each local demon then updates the 'life' and 'usage' totals for his box so that with these new values and the new value of *target* he can make a new decision if his box is entered during the next run.

The rules for updating and resetting adopted in the present case were as follows, using the convention that, say, X' means 'the previous value of X':

Consider a box in which the decision setting, S, was *left*, during the run which has just terminated.

Let N = the number of entries to the box during the run;
DK = constant multiplier less than unity which performs the function of weighting recent experience relative to earlier experience;
K = another multiplier weighting global relative to local experience;

then the local demon updates his totals, defined under (1) above, using:

$$LL = LL' \times DK + \sum_{i=1}^{N} (T_F - T_i);$$

$$LU = LU' \times DK + N;$$

$$RL = RL' \times DK;$$

$$RU = RU' \times DK.$$

The global demon has similar totals, 'global life', GL, and 'global usage', GU, and updates them using:

$$GL = GL' \times DK + T_F;$$
$$GU = GU' \times DK + 1;$$

From this the demon calculates *merit* as $\frac{GL}{GU}$,

and then computes *target* as $C0 + C1 \times merit$,
where $C0 \geqslant 0$ and $C1 \geqslant 1$.

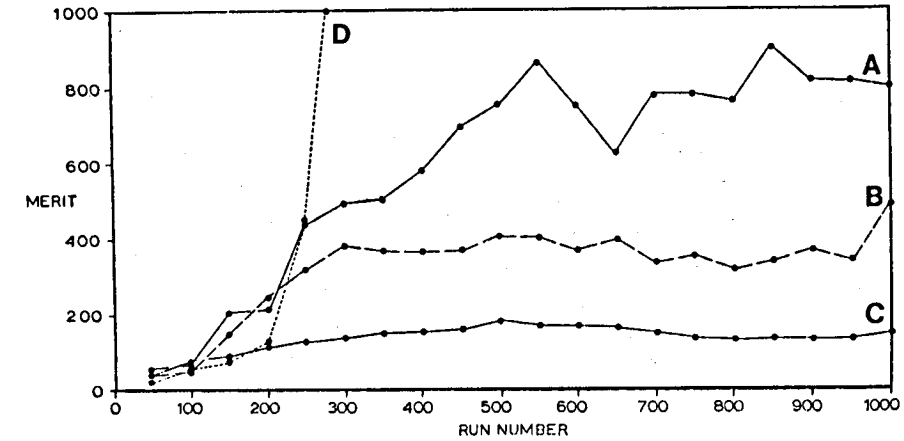
From *target* and his local totals, the local demon calculates the 'left value' of his box from

$$value_L = \frac{LL + K \times target}{LU + K}$$

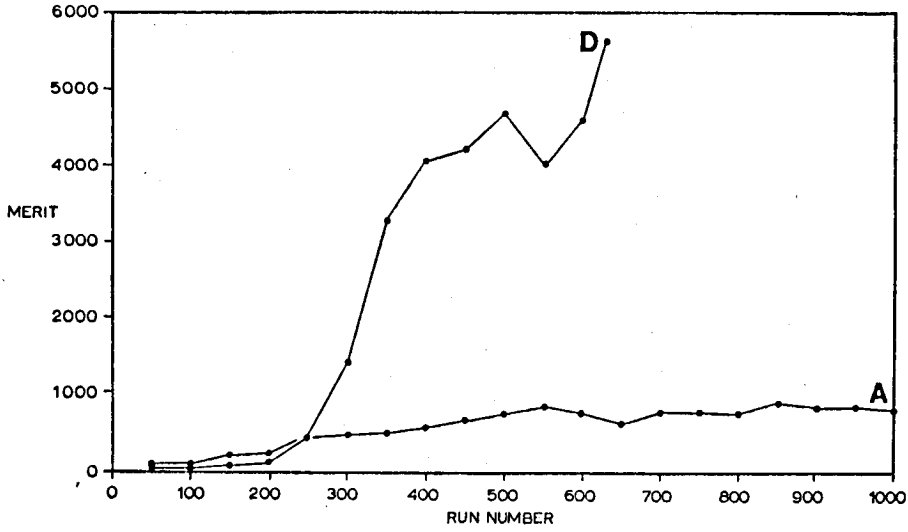
and similarly the 'right value', $value_R$. The control action, S, is selected as *left* or *right* according as $value_L$ is greater or less than $value_R$.

PERFORMANCE TESTS

After some preliminary trials the adjustable parameters *DK* and *K* were set at 0·99 and 20·0 respectively for the tests. The values of *C0* and *C1* were 0 and 1 respectively, throughout all the trials. Time has not allowed



(*a*)



(*b*)

FIG. 9. Duplicate runs of BOXES. Run A is shown on both graphs to facilitate comparisons.

any systematic attempt at optimisation since the decision algorithm has only recently been developed to the stage reported here. A test run consisted in a succession of attempts by the program, each attempt being terminated by receipt of the failure signal. For each new attempt, or 'control run', the

system state was set up by choosing a random point from a uniform distribution over the central region of the space defined by $x = \pm 21{\cdot}0$ in, $\dot{x} = \pm 1{\cdot}0$ in/sec, $\theta = \pm 6{\cdot}0$ deg and $\dot{\theta} = \pm 6{\cdot}0$ deg/sec.

Figs. 9 (*a*) and (*b*) show duplicate test runs A, B, C and D. In these test runs the value of *merit* (explained in the previous section) was recorded after the update which followed the failure of each control run, and the graphs show these values for every fiftieth control run. For each test run the same values were kept for all the adjustable parameters, but different starting values were given to the pseudo-random number generators which were used for resolving control decisions with equal left and right values and for setting up the new starting conditions of the system for each control run.

The wide limits within which starting conditions could be generated caused considerable variation in the duration of control runs. This is not obvious from the individual graphs because of the pronounced smoothing effect of the weighted-average calculation by which *merit* is computed, but in fact the length of extreme runs was a factor of 10, or more, greater or less than *merit*. Thus test run C produced a run of 1368 decisions at a point where *merit* was only 146, and test D, at a point with a *merit* value of 4865, produced a run of over 72 000 decisions which is equivalent to an hour of real-time control.

The use of different random number sequences in test runs, combined with this variation in the duration of control runs, means that the sequences of boxes entered by the system and the early decisions taken in those boxes differed greatly for each test run. Since the 'success' of any box depends on the decisions taken in neighbouring boxes, it is easy to see how the early decision settings for the whole collection of boxes were different for different test runs, whereas the low level of experience caused the control runs to be short and thus produce similar values of *merit* for all the tests. However, because of premature decision-taking fixing some of these early decision settings, the later behaviour of the tests varied considerably as can be seen in the graphs of Fig. 9, but the influence of the *target* term in the decision rule is noticeable even without optimisation. Graphs B and C are comparable, respectively, to the best and average performances of earlier, non-*target*, versions of BOXES. Graph A is better than any previous performance and D indicates what we hope to achieve when the target method has been improved.

'LEVEL OF ASPIRATION'

Earlier in this paper we referred to the fact that a move which on the evidence is disadvantageous may none the less be worth making if the expected information-gain associated with it is sufficiently high, and we emphasised that this fact should be reflected in the design of a trial-and-error automaton.

The problem of optimal information trade-off, as stated earlier, still awaits solution even in the simplest case—the 'two-armed bandit' problem. We have, in the meantime, evolved an approximate solution to this family of problems, and have found that it shows promise of being able to keep the 'premature

fixation' hazard at bay. It can moreover be made to yield, in some contexts at least, a rather close approach to optimality as we have verified by comparison with a limited range of optimal values computed for the two-armed bandit problem. The essential idea is to use a 'target' level of performance to calculate for the available alternative moves *not* their expected payoffs but 'optimistic' values of these expectations. Choice between moves is then made by comparing the optimistic values. These are calculated by taking a weighted mean between the expected value of the move and the target value. The weight of the expected value is taken as the amount of information associated with it, while the weight of the target value is an adjustable parameter of the program. The higher the target value and the greater its weight, the more 'research minded' is the behaviour of the automaton, tending to try new possibilities rather than to consolidate partial successes. Its 'level of aspiration', in anthropomorphic terms, is higher.

The idea is readily generalised, and can be immediately applied, for example to the GLEE automaton, or to the two-armed bandit itself. We have only recently started to explore this line: even the results of Fig. 9 were obtained before any optimisation of the parameters of the target method was attempted. We can, however, say in summary that the target method, when grafted on to the 'boxes' approach, has given good performance for adaptive control problems inaccessible to standard procedures.

REFERENCES

Donaldson, P. E. K. (1960). Error decorrelation: a technique for matching a class of functions. *Proc. III International Conf. on Medical Electronics.* pp. 173-178.

Michie, D. (1961). Trial and error, in *Science Survey, 1961* (eds. S. A. Barnett and A. McLaren) part 2, pp. 129-145. Harmondsworth: Penguin.

Michie, D. (1963). Experiments on the mechanisation of game-learning. 1. Characterisation of the model and its parameters. *Computer Journal*, 6, 232-236.

Newell, A., Shaw, J. C., & Simon, H. A. (1960). A variety of intelligent learning, in a general problem solver in *Self-organizing Systems* (eds. Marshall C. Yovits and Scott Cameron) pp. 153-189. London: Pergamon.

Schaefer, J. F., & Cannon, R. H. Jr. (1966). On the control of unstable mechanical systems. *Proc. IFAC 1966.* Paper 6C.

Selfridge, O. G. (1959). Pandemonium: a paradigm of learning, in *Mechanization of Thought Processes*, Vol. 1. N.P.L. Symposium No. 10, pp. 511-531.

Sworder, D. D. (1966). *Optimal Adaptive Control Systems.* Academic Press.

Widrow, B., & Smith, F. W. (1964). Pattern recognising control systems, in *Computer and Information Sciences* (eds. J. T. Tou and R. H. Wilcox). Clever Hume Press.

10

A REGRESSION ANALYSIS PROGRAM INCORPORATING HEURISTIC TERM SELECTION

J. S. COLLINS
EXPERIMENTAL PROGRAMMING UNIT
UNIVERSITY OF EDINBURGH

INTRODUCTION

Consider a system in which the value of one variable (called the *dependent variable*) depends on the values of one or more other variables (called the *independent variables*). An example of such a system is a furnace whose heat output depends upon the rates at which oil and air enter the system. If we measure the values of the heat output for different values of the oil and air flows, the resulting data will tell us something of the behaviour of the system. Regression analysis provides a method by which a formula which predicts the value of the dependent variable in terms of the values of the independent variables may be determined from such data. Thus, in the case of the furnace, the regression equation would give us a formula for calculating the heat output for any given values of oil flow and air flow.

The theory of regression analysis and the development of the standard formulae used is available in many standard statistics textbooks such as Kendall and Stuart (1961).

The basic data for regression analysis consists of a number of *observations*. An observation consists of the set of values of the $n+1$ system variables. There must be at least $n+1$ observations. The number of observations is normally large compared with the number of variables. This redundancy enables random errors in the observations to be smoothed out. The difference between the number of observations and the number of variables (a measure of the redundancy of the measurements) is called the number of *degrees of freedom*.

The result of a linear regression analysis on $n+1$ variables is a *linear regression equation* which predicts the value of the dependent variable as a function of the n independent variables. Let x_0 denote the dependent variable and $x_1, x_2, \ldots, x_n$ denote the n independent variables. A linear regression equation is of the form

$$y = b_0 + b_1 x_1 + \ldots + b_n x_n$$

where y is the predicted value of the dependent variable and $b_0, b_1, \ldots, b_n$ are constants known as the *regression coefficients*. The values of the regression coefficients are computed to minimise $\Sigma(y-x_0)^2$ where the sum is over all observations. This method successfully smooths small random errors in the data if the order of magnitude of the errors is approximately the same for each observation. The existence of one large error in the data is likely to invalidate the regression equation. This happens because the differences between actual and predicted values of the dependent variable are squared.

A FORTRAN program for carrying out linear regression analysis is described by Dallemand (1958). The analysis is carried out in a way which makes the program particularly useful when only a subset of the set of independent variables significantly affects the behaviour of the dependent variable. The regression equation is built up one variable at a time starting with a constant term. The variable added at each stage is that independent variable which best predicts the discrepancy between the actual behaviour of the dependent variable and the prediction of the regression equation so far. If the best available independent variable does not significantly improve the accuracy of the regression equation, no variable is added and the assembly process stops. If at any time during the assembly process, a variable in the regression equation ceases to contribute significantly to the accuracy of the regression equation, that variable is deleted from the equation. The final regression equation is such that all the variables in the equation are significant and all the variables not in the equation are not significant.

Dallemand also describes an extension to the regression analysis program which allows non-linear regression equations to be produced. By computing functions of each of the values of the independent variables for each observation in the data and presenting the result to the linear regression analysis as if it were raw data, more complex relationships can be studied. Functions commonly used for this purpose are powers, reciprocals or logarithms of the independent variables.

Westervelt (1960) describes a further extension to this type of regression analysis. The main features of this development are described in the remainder of this section.

The regression equation consists of a linear combination of a number of *terms*. Each term consists of one or more *factors* and the factors are simple functions of the independent variables. The number of factors in a term is called the *interaction order* of the term. If there are n independent variables and p functions then each independent variable may be either not included or

included in a term as one of the p functions. There are thus $p+1$ possibilities for each independent variable and therefore $(p+1)^n$ combinations in all. This however includes the term in which all factors are absent and therefore the total number of terms is $(p+1)^n-1$. This number increases rapidly with p and n. Even for a relatively simple problem involving 10 functions of 5 independent variables, the total number of terms is 161 050.

Such large numbers of terms cannot be treated by a conventional regression analysis procedure which needs to store a correlation matrix which involves every pair of terms. It is therefore necessary to examine the terms a few at a time in a search procedure. The solution adopted was to try a small number of terms during each of a number of trials, retaining those terms found to be significant by the regression analysis for the following trial. This alone was not sufficient to handle the large number of possible terms. The selection of terms for each trial was made to depend upon the significance or otherwise of similar terms tested in previous trials. Thus the results of each trial help to direct the search for suitable terms into apparently promising areas of the term space.

Similarity between terms is defined by a classification scheme. Terms are classified according to the number of factors and for each factor, the independent variable concerned and the function used. As a result of previous trials, the program learns which interaction orders (number of factors in a term), independent variables and functions are promising. Westervelt gave no indication of the effectiveness of this learning mechanism.

A GENERAL REGRESSION PROGRAM

Using the schemes described by Westervelt and Dallemand, a general regression program was written. The object was to investigate the efficiency of the search process in an improved search space, and at the same time produce a useful program.

A simplified flow diagram of this general regression program is shown in Fig. 1. On entry to the program, standard initial conditions are set up. Apart from essential data, the problem specification need only contain desired departures from these standard conditions. An annotated example of data prepared for the regression program is given in Appendix 1. A set of terms is selected which are evaluated for each observation in the data. The resulting edited data are analysed by the regression program resulting in a regression equation containing those terms which are found to be significant.

If the regression equation produced satisfies the accuracy criterion specified by the user, the details of the regression equation are printed and the process is terminated. If the resulting regression equation is not sufficiently accurate, a further trial is initiated. Before a new set of terms is selected, the term selector is modified to encourage the selection of terms similar to those found to be significant in the previous trial. These terms are retained and appear in the set of terms selected for the next trial. Terms similar to those not found significant in the previous trial are discouraged. These terms are replaced by new terms for the following trial.

The most important parts of the program in respect of the investigation were the term selector and term selector modifier. Because of this and because the other parts of the program use well-known techniques which are described elsewhere, only these parts of the program are described in detail.

A term is a product of a number of factors. A factor may be one of four types of function of one of the independent variables. The four types are positive powers, x, x^2, x^3, . . . , reciprocal powers x^{-1}, x^{-2}, x^{-3}, . . . , roots $x^{1/2}$, $x^{1/3}$, . . . and reciprocal roots $x^{-1/2}$, $x^{-1/3}$, . . . These four function types are referred to as types 0, 1, 2 and 3 respectively. The *order* of a function

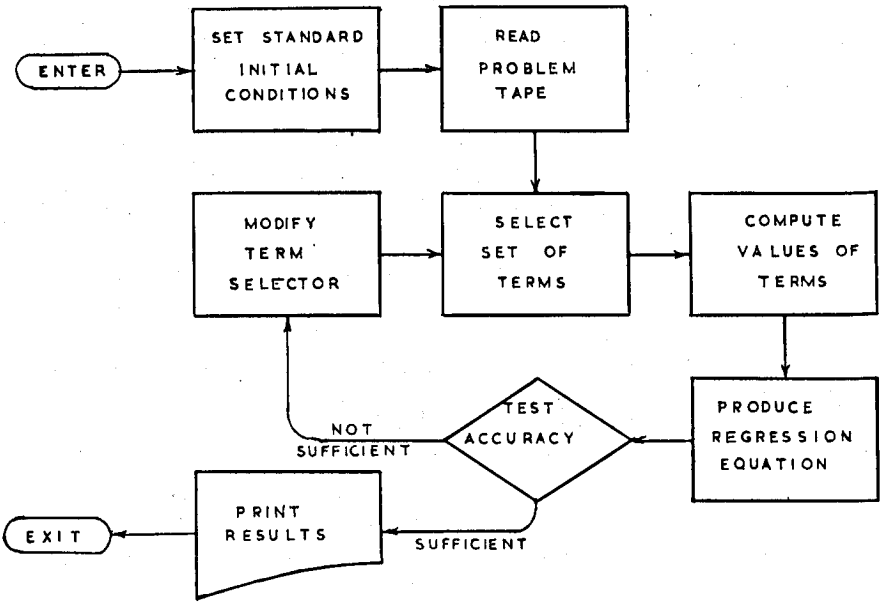


FIG. 1. Flow diagram of the regression program.

defines which of the series of functions corresponding to each function type is required. For example x^2, x^{-2}, $x^{1/2}$ and $x^{-1/2}$ are all functions of order 2.

A factor is selected with the aid of a *factor selection tree*. The factor selection tree for three independent variables and a maximum function order of three is shown in Fig. 2. A factor is selected by choosing a path in the tree from the starting node to one of the terminal nodes. Each terminal node corresponds to one distinct factor. Three consecutive choices are required to select one factor. The first choice is between the three independent variables. The second choice is between the four function types and the third choice is between the possible orders for the chosen function type. Functions of type 2 and 3 have a minimum order of 2 as an order of 1 would duplicate functions included in types 0 and 1.

The order in which the choices are made defines the manner in which all

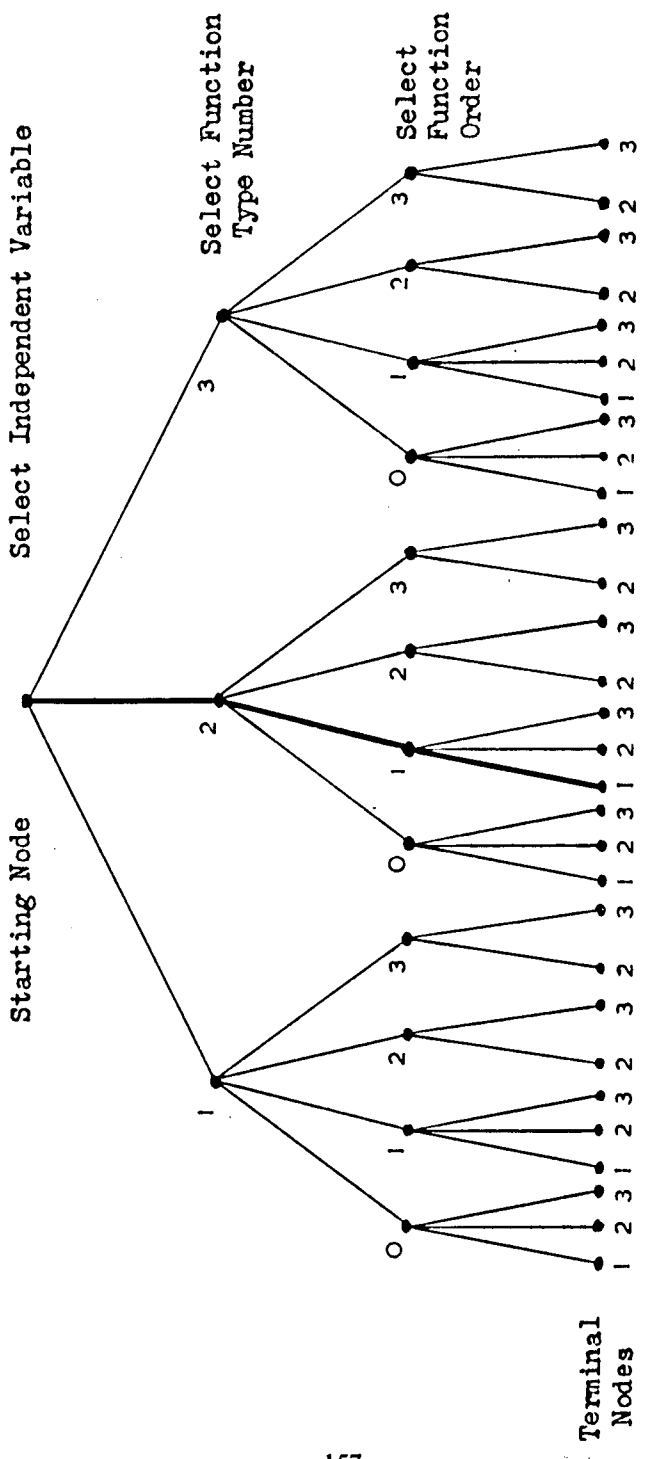


FIG. 2. The factor selection tree.

possible factors are classified. The heavy line in Fig. 2 indicates the choice path which results in the selection of the factor x_2^{-1}. Let a probability be associated with the choice of every possible branch at each decision node. There are therefore three probabilities at the starting node which indicate the relative probabiliity of selecting each possible independent variable for a factor. The selection probability of any factor is equal to the product of the three probabilities associated with the three branches of the selection path defining the factor.

Let the three probabilities involved in the choice of the factor x_2^{-1} be each multiplied by *probability multipliers a, b* and *c* respectively. Not only does this affect the probability of selecting x_2^{-1}, but the selection probabilities of other factors are altered also. The selection probabilities of the factors x_2^{-2} and x_2^{-3} are multiplied by *ab* and the selection probabilities of x_2, x_2^2 and x_2^3 are multiplied by *a*. Thus the effect of modifying the selection probability of one factor is to modify, by different amounts, the selection probabilities of other factors which are defined as similar to the given factor by the structure of the factor selection tree. The values of the probability multipliers control the degree of this effect on similar factors.

By reducing any probability to zero, the selection probability of any class of factors may be reduced to zero. This is useful, for example, to eliminate the selection of reciprocals when zeros exist in the observations.

A term is characterised by the factors it contains and the number of them. The order of factors in a term is immaterial. In order to select a term, it is first necessary to decide upon the interaction order (number of factors) of the term. The factor selection tree is then used once to select each factor of the term. The selection of further factors containing a previously selected independent variable is avoided by temporarily setting the independent variable probability to zero after the selection of each factor. There is a probability associated with each possible interaction order.

Terms are selected using a random number generator which selects between possibilities with the given probabilities. When a set of terms has been selected and a regression equation has been formed from a subset of them, the interaction order probabilities and factor selection tree probabilities are modified to encourage the selection of promising terms. The probabilities involved in the selection of terms in the regression equation are multiplied by probability multipliers greater than unity. The probabilities involved in the selection of terms not in the regression equation are multiplied by probability multipliers less than unity. Because all probabilities are multiplied by finite non-zero probability multipliers, the selection probability of no factor can ever become zero. All possible terms therefore always have a finite probability of being selected.

The choice of values for the probability multipliers naturally affects the efficiency of the search procedure. If their values are too near to unity, the search is slow to take advantage of new discoveries. If their values are too far removed from unity, the search is too erratic.

PERFORMANCE OF THE PROGRAM

A regression program, using the techniques described in the previous section has been written in Mercury Autocode for the Ferranti Mercury computer. The program can deal with up to 200 observations on up to 15 variables. In each trial of the analysis, up to 15 terms can be selected, evaluated and tested for significance by the regression algorithm.

The program has been tested on data from a variety of sources. This includes actual physical data as well as artificially generated data. Physical data analysed by the program has included ecological, engineering and simple calibration measurements. In some cases, suitable formulae have been produced by the program. In others, the significant terms discovered by the program have aided subsequent theoretical analysis.

One application of this program described elsewhere (Collins 1964) has been to assist the programmer in coding subroutines whose argument list/value relationship is known by example but not as an explicit function. The application was concerned with a search procedure and the subroutine in question was required to decide whether to proceed along a certain path or to give up. The parameters of this subroutine were a set of variables on which the choice was thought to depend. The program was run using a 'human subroutine' (arguments displayed on an on-line typewriter and value typed in by the user) and the argument list/value pairs were collected and later analysed by the regression program. The formula produced by the regression program was then placed in the search program in place of the human subroutine. The new search program was then run and its performance compared with that using the human subroutine.

From the point of view of testing the performance of the program, the most valuable tests have been on data generated by the evaluation of analytic expressions for a number of sets of values of the independent variables. By selecting expressions which can be represented exactly by the general regression program, the progress of the search towards the known goal can be observed. A test of this type is fully documented in Appendix 2.

As a result of tests with different values for the probability multipliers, the following values were found to be generally suitable.

Interaction Order:	2·0	0·50
Independent Variable:	1·5	0·67
Function Type:	1·5	0·67
Function Order:	1·25	0·80

The first probability multiplier is for terms included in the regression equation; the second which is the reciprocal of the first is for terms not included in the regression equation. The values are not critical. The function order multiplier should be closer to unity than the function type multiplier to prevent emphasis of a particular function order rather than type. Using these probability multipliers, the selection probability of a factor is multiplied or

divided by 2·8125 as a result of its significance or non-significance in the regression analysis. Similarly, the selection probability of a term is multiplied or divided by $2 \times 2{\cdot}8125^n$ where n is the number of factors in the term, and the selection probability of a similar term in which one factor is different is multiplied or divided by $2 \times 2{\cdot}8125^{(n-1)}$.

CONCLUDING REMARKS

A computer program for generating analytic expressions to explain experimental measurements has been described. The use of heuristics enables the program to find suitable expressions in a much shorter time than would otherwise be possible.

In some cases a theoretical justification has been found for terms produced by the program from experimental measurements. In many other cases, the program has suggested terms which have provided clues helpful in the theoretical analysis of the experimental system.

The use of a selection tree, in which selection is controlled by probabilities, has been found to be a workable method of defining similarity between discrete objects. A selection tree defines a space of all possible objects in which similar objects are closely situated. The method is analogous to the classification of books in a library in which similar books occupy neighbouring positions on the shelves. The result of this type of classification is that much less effort is required to find a book on a given subject than if the books were placed at random. Tests on the regression program have shown the classification method used to be satisfactory. There is some tendency for multiple peaks to exist in the search space but the use of random numbers for term selection prevents the search process from getting stuck at these points.

A possible modification of the factor classification scheme suitable for the current set of functions could be to classify positive powers with positive roots and negative powers with negative roots. If this is done, the significance of a factor such as $x_1^{1/2}$ increases the selection probability of a factor such as x_1. The resulting modified factor selection tree is shown in Fig. 3. This suggestion was not implemented as it is function dependent. That is, it works for the particular functions used but not if other functions are used. It was considered desirable in this application to keep the program function independent. Because of this it is possible to use the program with any other functions simply by changing the routines responsible for calculating the values of the functions in terms of their symbolic description and for printing a description of the terms.

A noticeable property of the program is that the change in the set of terms from trial to trial behaves somewhat like a simple evolutionary system. Terms found by the regression analysis to be significant 'survive' for use in the next trial. In addition the probability of a similar term being generated is increased. Consider a particular characteristic of a term such as its interaction order. Let a and b be the interaction order probability multipliers for significant and non-significant terms respectively. For each significant

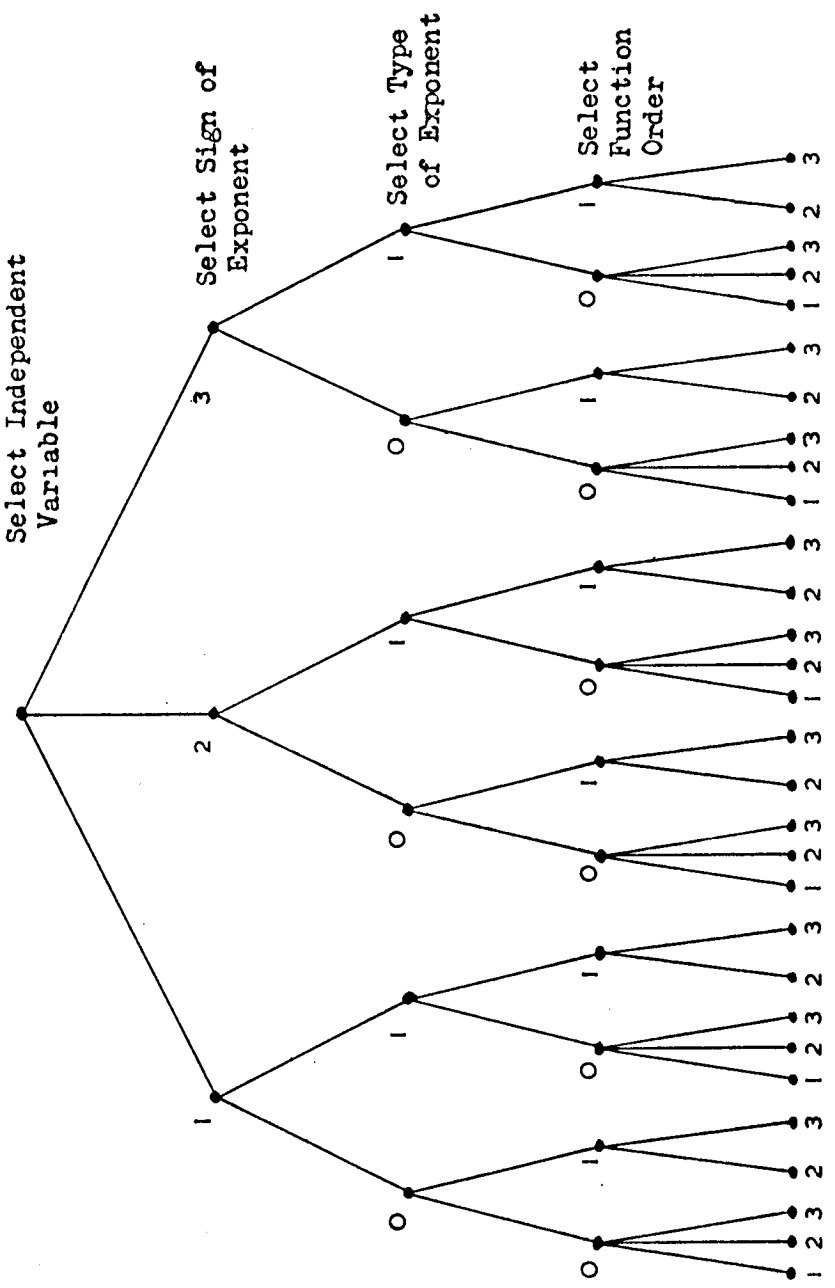


FIG. 3. Modified factor selection tree.

term with a given interaction order suppose there are n non-significant terms with a similar interaction order. When the term selector is modified, the selection probability for the given interaction order is multiplied by ab^n. For steady conditions $ab^n = 1$ and hence

$$n = -\log(a)/\log(b)$$

If $b = 1/a$ then $n = 1$. This applies to the general regression program. Each significant term will on average be balanced by a non-significant term with the same interaction order. This property applies to other features of the terms as well as the interaction order. Thus the region surrounding the current area of interest is explored. If the number of significant terms is less than half the number of terms selected, there will be, on average, some extra terms selected with random properties. These terms explore the term space at greater distances from the current area of interest and thus help the search process to overcome the problem of multiple peaks.

ACKNOWLEDGEMENTS

The author is indebted to British Petroleum Limited for providing generous amounts of time on their Mercury computer for the development of the program described. Thanks are also due to Professor M. W. Humphrey Davies for his advice and encouragement while supervising the work.

REFERENCES

Collins, J. S. (1964). 'The processing of lists and the recognition of patterns with application to some electrical engineering systems', Ph.D. Thesis, University of London.

Dallemand, J. E. (1958). Stepwise regression program on the IBM 704, *General Motors Research Staff Report, GMR 199.*

Kendall, M. G., & Stuart, A. (1961). *The Advanced Theory of Statistics*, Vol. 2. London: Griffin.

Westervelt, F. H. (1960). 'Automatic System Simulation Programming', Ph.D. Thesis, University of Michigan.

APPENDIX 1

The following is an annotated example of some data prepared for the regression program.

Information Punched	Notes
TITLE J362, COLLINS, EXAMPLE A.	The line of text following TITLE is printed at the head of the results for identification.
VARIABLES=3	This problem involves three independent variables.
FUNCTION RANGES	Terms are made up from the following factors:
3 3 0 0	$x_1\ x_1^2\ x_1^3\ 1/x_1\ 1/x_1^2\ 1/x_1^3$
3 3 3 3	$x_2\ x_2^2\ x_2^3\ 1/x_2\ 1/x_2^2\ 1/x_2^3\ x_2^{\frac{1}{2}}\ x_2^{\frac{1}{3}}\ 1/x_2^{\frac{1}{2}}\ 1/x_2^{\frac{1}{3}}$
3 0 3 0	$x_3\ x_3^2\ x_3^3\ x_3^{\frac{1}{2}}\ x_3^{\frac{1}{3}}$
INTERACTION LIMIT=2	Only terms made up of up to 2 factors are to be selected.
PRINTING REQUIRED TRIALS EACH REGRESSION STEP	Terms generated for each trial and each step of the regression analysis are to be printed.
READ UNWEIGHTED DATA 1·000 22·5 1·21 368 2·450 24·5 1·30 924 1·414 23·5 1·47 1145 1·000 22·5 1·47 1120 2·646 22·5 1·47 1161 2·646 21·5 1·55 3124 1·732 24·5 1·59 920 2·450 22·5 1·67 444 1·732 22·5 1·43 671 0 22·5 1·50 0	This is the actual data on which the regression analysis is carried out. Each observation is set across a line consisting of the value of the dependent variables followed by the values of the three independent variables.
*	An asterisk terminates the data.

Information Punched	Notes
INITIAL TERMS 1 3 1 0 2 1 2 0 2 1 1 *	The user may suggest a set of terms to be included in the first trial. In this case the terms x_3 and x_1^2/x_2 are suggested. An asterisk terminates the list of initial terms.
ACCURACY = 0·001	Indicates the accuracy at which analysis should stop.
TRIALS LIMIT = 20	Stop anyway after 20 trials.
BEGIN	When read this causes analysis to begin.

APPENDIX 2

The following is a summary of the performance of the search procedure used in the regression program. Artificial data was generated by evaluating the expression

$$1+2x_1-3\frac{x_1 \cdot x_3^3}{x_2}+x_4^2$$

for eighty sets of values of the independent variables (x_1, x_2, x_3 and x_4) chosen randomly in the range 0 to 10. The values of both the expression and the independent variables were punched on paper tape to an accuracy of three decimal places in a form suitable for use with the regression program. The following summary shows how the program searched through a large number of possible terms and found the original expression from which the data was generated.

Terms included in the search were made up of factors consisting of powers, reciprocals of powers, roots and reciprocals of roots each up to order 5. This allows 18 possible functions of each variable.

For each trial, 15 terms were selected, evaluated and tested by the regression procedure. Below are shown those terms in each trial which were found to contribute significantly in a regression equation. In each case, the standard error (the root mean square of the deviations between actual and predicted values) of the regression equation is given.

The terms found to be significant in one trial are always automatically included among the set of terms tested in the following trial. Such terms are marked with an asterisk in the commentary below.

Trial No.	Terms included in regression equation	St. error of Y	Notes
0	NONE	1254·5	The standard error in this case is the standard deviation of x_0 about the mean.
1	$\frac{x_1^{\frac{1}{2}} \cdot x_3^2 \cdot x_4^2}{x_2^{\frac{1}{2}}}$ $x_1^{\frac{1}{2}} \cdot x_3^3$ $x_1 \cdot x_2^{\frac{1}{2}} \cdot x_3^3 \cdot x_4^{\frac{1}{2}}$	754·30	Two out of the three terms found significant include the factor x_3^3.
2	$*x_1^{\frac{1}{2}} \cdot x_3^3$ $\frac{x_1^{\frac{1}{2}} \cdot x_3 \cdot x_4^2}{x_2}$ $\frac{x_1^{\frac{1}{2}} \cdot x_3^3}{x_2^3 \cdot x_4^{\frac{1}{2}}}$	645·54	The factor $1/x_2$ appears for the first time.

Trial No.	Terms included in regression equation	St. error of Y	Notes
3	$*x_1^{\frac{1}{2}} \cdot x_3^3 \quad \frac{x_1^{\frac{1}{2}} \cdot x_3}{x_2 \cdot x_4^{\frac{1}{2}}} \quad \frac{x_1^{\frac{1}{2}} \cdot x_3^5}{x_2^4}$ $\frac{x_1^{\frac{1}{2}} \cdot x_3^3 \cdot x_4^{\frac{1}{2}}}{x_2} \quad x_1^{\frac{1}{2}} \cdot x_3^3$	280·0	The fourth term contains both the factors x_3^3 and $1/x_2$.
4	$\frac{*x_1^{\frac{1}{2}} \cdot x_3^5}{x_2^4} \quad \frac{*x_1^{\frac{1}{2}} \cdot x_3^3 \cdot x_4^{\frac{1}{2}}}{x_2}$ $*x_1^{\frac{1}{2}} \cdot x_3^3 \quad \frac{x_1^{\frac{1}{2}} \cdot x_3^3}{x_4^{\frac{1}{2}}} \quad \frac{x_3^4}{x_1^{\frac{1}{2}} \cdot x_2}$ $\frac{x_3^3}{x_2^{\frac{1}{2}}} \quad x_1^{\frac{1}{2}} \cdot x_3^3$	154·19	
5	$\frac{*x_1^{\frac{1}{2}} \cdot x_3^3 \cdot x_4^{\frac{1}{2}}}{x_2} \quad * \frac{x_3^4}{x_1^{\frac{1}{2}} \cdot x_2}$ $\frac{x_3^3 \cdot x_4^{\frac{1}{2}}}{x_2} \quad \frac{x_3^5}{x_1 \cdot x_2}$ $\frac{x_1^5 \cdot x_3^3}{x_2} \quad \frac{x_1^{\frac{1}{2}} \cdot x_3^3}{x_2}$	54·076	Three terms are near misses here including the factors x_3^3, $1/x_2$ and a positive power or root of x_1.
6	$* \frac{x_3^4}{x_1^{\frac{1}{2}} \cdot x_2} \quad \frac{*x_3^3 \cdot x_4^{\frac{1}{2}}}{x_2}$ $* \frac{x_3^5}{x_1 \cdot x_2} \quad \frac{*x_1^5 \cdot x_3^3}{x_2}$ $\frac{*x_1^{\frac{1}{2}} \cdot x_3^3}{x_2} \quad \frac{x_1^2 \cdot x_3^3}{x_2}$	31·864	Note how terms 4, 5 and 6 are 'converging' on one of the terms in the original equations.
7	$\frac{x_4^4}{x_2} \quad \frac{x_1 \cdot x_3^3}{x_2} \quad \frac{x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}}}{x_2}$	20·105	The second term is one of the three terms in the original equation.
8	$\frac{*x_1 \cdot x_3^3}{x_2} \quad \frac{x_1}{x_4^{\frac{1}{2}}} \quad \frac{x_3^{\frac{1}{2}}}{x_1^2}$ $\frac{x_3^{\frac{1}{2}} \cdot x_4^5}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}} \quad x_1$	10·900	A second term of the original equation (x_1) is now found.

Trial No.	Terms included in regression equation	St. error of Y	Notes
9	$*\frac{x_1 \cdot x_3^3}{x_2}$ $*\frac{x_3^{\frac{1}{2}}}{x_1^2}$ $*\frac{x_3^{\frac{1}{2}} \cdot x_4^5}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $\frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4}{x_2^{\frac{1}{2}}}$	9·1994	x_1 is dropped out due to lack of significance.
10	$*\frac{x_1 \cdot x_3^3}{x_2}$ $*\frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4}{x_2^{\frac{1}{2}}}$ $\frac{x_3^{\frac{1}{2}} \cdot x_4^{\frac{1}{2}}}{x_1^2}$ $\frac{x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $\frac{x_4^4}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $\frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2}$ $1/x_1^2$	5·0410	Positive powers of x_4 are beginning to appear.
11	$*\frac{x_1 \cdot x_3^3}{x_2}$ $*\frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4}{x_2^{\frac{1}{2}}}$ $\frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $\frac{*x_4^4}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $*\frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2}$ $\frac{x_4^4}{x_1^2 \cdot x_2}$ x_1^2 $\frac{x_2^{\frac{1}{2}} \cdot x_4^5}{x_1^2}$	4·3066	
12	$*\frac{x_1 \cdot x_3^3}{x_2}$ $*\frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4}{x_2^{\frac{1}{2}}}$ $\frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $*\frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2}$ $\frac{*x_4^4}{x_1^2 \cdot x_2}$ $*x_1^2$ $*\frac{x_2^{\frac{1}{2}} \cdot x_4^5}{x_1^2}$ $\frac{x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2^{\frac{1}{2}}}$	3·5597	

Trial No.	Terms included in regression equation	St. error of Y	Notes
13	$*\frac{x_1 \cdot x_3^3}{x_2}$ $*\frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4}{x_2^{\frac{1}{2}}}$ $\frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $*\frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2}$ $*x_1^2$ $*\frac{x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2^{\frac{1}{2}}}$ $\frac{x_4^5}{x_1^2 \cdot x_2 \cdot x_3^{\frac{1}{2}}}$	3·5882	is slightly worse than that of previous trial but there is one less term in the equation.
14	$*\frac{x_1 \cdot x_3^3}{x_2}$ $\frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $*x_1^2$ $*\frac{x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2^{\frac{1}{2}}}$ $\frac{*x_4^5}{x_1^2 \cdot x_2 \cdot x_3^{\frac{1}{2}}}$ $\frac{x_3^{\frac{1}{2}} \cdot x_4^5}{x_1^{\frac{1}{2}} \cdot x_2}$	3·5365	Again, one less term than in the previous trial but increased average significance of each term.
15	$*\frac{x_1 \cdot x_3^3}{x_2}$ $\frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $*x_1^2$ $*\frac{x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2^{\frac{1}{2}}}$ $*\frac{x_3^{\frac{1}{2}} \cdot x_4^5}{x_1^{\frac{1}{2}} \cdot x_2}$ $1/x_1^{\frac{1}{2}}$ $\frac{x_4^2}{x_1^2 \cdot x_2 \cdot x_3^{\frac{1}{2}}}$	2·5800	Two terms now include x_4^2.
16	$*\frac{x_1 \cdot x_3^3}{x_2}$ $\frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $*x_1^2$ $*\frac{x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2^{\frac{1}{2}}}$ $*\frac{x_3^{\frac{1}{2}} \cdot x_4^5}{x_1^{\frac{1}{2}} \cdot x_2}$ $\frac{*x_4^2}{x_1^2 \cdot x_2 \cdot x_3^{\frac{1}{2}}}$ $x_1^{\frac{1}{2}}$	2·5001	
17	Same as 16		

Trial No.	Terms included in regression equation	St. error of Y	Notes
18	$*\frac{x_1 \cdot x_3^3}{x_2}$ $*x_1^2$ $*x_1^{\frac{1}{2}}$ x_4^2 $\frac{x_3^{\frac{1}{2}}}{x_1^2 \cdot x_2}$	0·5302	The term x_4^2 of the original equation has now been found.
19	$*\frac{x_1 \cdot x_3^3}{x_2}$ $*x_4^2$ x_1	0·3635	All the original terms are found. Analysis is complete. The standard error is not zero because of small random errors introduced into the original data by truncating each figure to 3 figures accuracy.

A graph of the logarithm of the standard error against the trial number is shown in Fig. 4. The graph shows the steady improvement of accuracy obtained by the process. This is a useful feature when the best regression equation for a given computational effort is required.

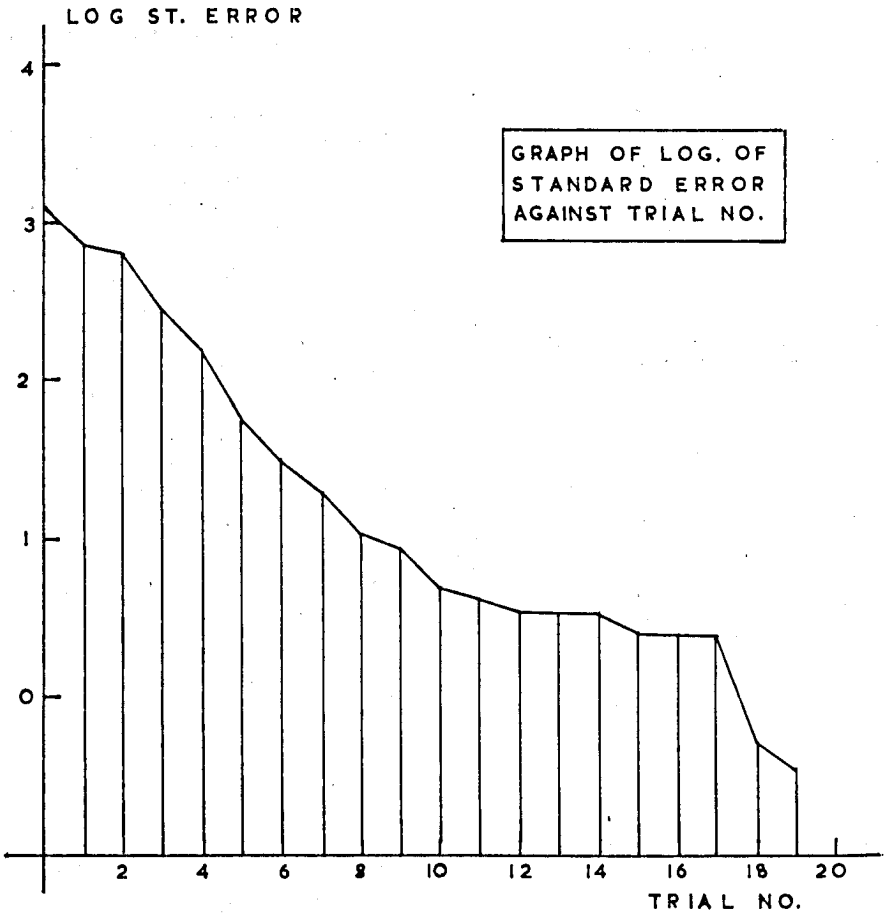


FIG. 4. The progressive improvement of the regression equation.

The analysis described above required approximately 45 minutes on the Ferranti Mercury computer, during which only 181 out of 130 320 possible terms were tested.

COGNITIVE PROCESSES: METHODS AND MODELS

11

A LIMITED DICTIONARY FOR SYNTACTIC ANALYSIS

P. BRATLEY and D. J. DAKIN *
DEPARTMENT OF COMPUTER SCIENCF AND ENGLISH LANGUAGE RESEARCH UNIT
UNIVERSITY OF EDINBURGH

INTRODUCTION

Work is at present proceeding at the English Language Research Unit of the University of Edinburgh on the construction of a computer model for the perception of syntactic structure (Thorne *et al.* 1966). One of the basic principles behind the construction of the model is that it should be possible to analyse sentences without recourse to a complete dictionary of the language under consideration (Thorne 1964). However, it is not possible to dispense with a dictionary completely, as certain words (such as, for instance, determiners, prepositions and so on) play a role of major importance in the syntax. These words, which we call closed-class words, must be listed in a closed-class dictionary, and it is with the production of this dictionary that the present paper is concerned.

Consider for a moment the well-known lines:

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe.'

It is hard to maintain that the lines actually mean anything even if one knows that *toves*, for instance, are little animals that live under sundials (or is that borogroves?). Nevertheless, it is not at all unreasonable to assert that the lines have a readily perceived syntactic structure. In everyday terms, we might say that *slithy* is clearly an adjective, *tove* is a noun, and *gyre* and *gimble* are verbs.

* The work described is supported in part by a DSIR grant, contract number ID/102/2/60.

Notice what we are doing: we are not saying, 'I know that *tove* is a noun, therefore the syntactic structure is so-and-so.' On the contrary, we are saying, 'The syntactic structure is apparent, and knowing the structure, I can now assert that *tove* is a noun.'

On the other hand, suppose we are faced with the line:

William the the and disappointed.

No syntactic structure is now recognisable, even though all the words in the line are well known to us.

We maintain that the fact that the first example apparently has a syntactic structure, while the second example does not, is explained as follows. Certain words in English, for example *the*, *and* and *in*, carry a great deal of information about the syntactic structure of the sentences in which they occur. They are called closed-class words, on the grounds that there are, for instance, only a finite number of determiners in the language, and only a finite number of conjunctions. On the other hand, words such as *iron*, *book* or *green* play a very indeterminate part in the syntactic structure, so that *iron*, for instance, may equally well be a verb, a noun or an adjective. Words such as these are called open-class words, since one can imagine that the class of nouns, say, has an infinite number of members.

The syntactic structure of a sentence is determined by the closed-class words in that sentence, and the parts that the open-class words are playing on any particular occasion can be discovered from the structure thus determined. So the first example above is, syntactically at least, perfectly normal; but the second example has the closed-class words misused, so no structure is discoverable.

A computer model based on this principle of the distinction between open- and closed-class words has been developed at Edinburgh, and is now working on KDF9. Each word of an input sentence is looked up in a closed-class dictionary, to see if it is one of the words used to determine structure. If it is, it is coded appropriately. If not, the word is simply coded as an open-class item (although as described below some notice is taken of word-endings, so that 'open-class item ending with *-s*', for instance, is coded differently from a simple open-class item). For example, if the input sentence were

The old grey goose is dead,

the successive words of the sentence would be coded, roughly, as

DET O/C O/C O/C SING. VERB O/C.

Of this sentence, only the words *the* and *is* need appear in the dictionary.

We estimate that a closed-class dictionary for English, to be used in this way as part of a model for the perception of syntactic structure, would need to contain at most about 2000 items. As open-class words are not listed at all, however, it will be clear that the model will be capable of dealing with sentences from an infinite vocabulary: for example, new nouns or new verbs make no difference to its functioning whatsoever.

After this very brief introduction to the ideas of open-class and closed-class, and to the idea of a closed-class dictionary being used in a model for syntax recognition, the rest of the paper will deal with some of the decisions we have taken and the problems we have met in endeavouring to construct such a dictionary.

WORD-ENDINGS

Word-endings are of two types, traditionally known as *inflection* and *derivation*. Inflection refers, among other things, to the final *-s* of plural nouns, the possessive case endings, and the various forms of a verb. Derivation refers to the processes by which endings such as *-ise*, *-ment* and *-ish* are added to existing words to form new verbs, nouns and adjectives. The two kinds of ending do not seem to be equally important. For the analysis of the sentence,

(1) The boys fish,

information about the closed-class item, *the*, and about inflected word-endings is quite sufficient. The dictionary must contain entries to the effect that:

(*a*) *the* is a determiner which may be followed by a singular or a plural noun;

(*b*) an open-class item ending in *-s* may be the plural form of a noun or the singular form of a verb;

(*c*) an uninflected open-class item (i.e., one without any inflected ending) may be the singular form of a noun or the plural form of a verb. (For the sake of simplicity we temporarily ignore the fact that it may also be an adjective or the first person singular form of a verb.)

With this information the program can determine that *the boys* must be the subject of the sentence, that *boys* must be a plural noun, and that *fish* must be the verb. The same information will correctly permit two analyses for:

(2) The girl guides fish.

One analysis will treat *the girl guides* as the subject; the other will treat *the girl* as the subject and *guides* as a verb.

So far no information about derivational endings has been required. It is not clear that such information would ever be decisive. Let us consider the ending *-ment* as an illustration. Most words ending in *-ment* are nouns. If we were to include information to this effect in the dictionary, we would also have to include a list of exceptions like *vehement* which can never be nouns, and a longer list of words like *comment* and *torment* which can be either nouns or verbs. Lists of this kind are redundant in the case of sentences like:

(3) They torment fish,

(4) They were in torment.

Information about the closed-class items *they*, *in* and *were* is sufficient to determine whether *torment* is to be regarded as a noun or a verb. In the more critical case where information drawn from closed-class items and inflectional

endings cannot prevent an ambiguous analysis, derivational information is unlikely to remove the ambiguity. In this sentence, for example,

(5) The boy torments fish,

the derivational information that *torments* may be a noun or a verb reveals no more than what can already be discovered from the inflected ending *-s*. It cannot prevent an analysis of the sentence treating *the boy torments* as a subject of the same structure as *the girl guides* in one analysis of sentence (2).

Information about closed-class items and inflectional endings must clearly be included in the dictionary. So, of course, must exceptions to any mechanical routine for identifying inflected endings. Words like *lens* and *hatred* can scarcely be treated as the singular and past forms of particular verbs. Information about derivational endings, on the other hand, seems at present dispensable. There are, however, two exceptions. The endings *-ly* and *-able* cannot be ignored. If they were, *accidentally* would be treated as an uninflected noun in this sentence

(6) The boy accidentally slipped

on a par with

(7) The boy scout slipped.

The ending *-ly* is therefore included in the dictionary as an ending for adjectives and adverbs, and all exceptions are listed. *Accidentally* would not, of course, be treated as an adjective in sentence (6), since the program contains a general rule to the effect that adjectives do not follow nouns within the same noun phrase. The few possible exceptions to this rule like *opposite* are listed in the dictionary, and so are the derivational endings *-able* and *-ible*. *Available* must obviously be treated as an adjective in

(8) He saw all the men available

and not as an uninflected verb on a par with *die* in

(9) He saw all the men die.

CLOSED-CLASS ITEMS

Reference has already been made to relatively small classes of closed-class items such as pronouns, prepositions, and conjunctions. These must be exhaustively listed in the dictionary for two reasons. They must not be treated as uninflected forms of nouns and verbs. They also provide specific information which can be used to determine the structure of sentences. The information contained in traditional labels such as *pronoun*, *preposition* is not, however, specific enough. The word *her*, for instance, might be traditionally classified as a pronoun and as a possessive article. But it certainly could not be a pronoun in this sentence:

(10) Her dislikes increase as she grows older,

or a possessive article in this:

(11) I saw her.

To prevent such analyses being made, the dictionary must contain information about *her* which indicates that as a pronoun it cannot be the subject of a verb, and as a possessive article it must be followed by a noun. The functions of words traditionally known as pronouns, articles, possessives, determiners, and so on, are specified in the dictionary by code-words, which indicate whether, among other things, they

(*a*) may begin a subject noun phrase, e.g., you, she, someone, some, this, open-class items;
(*b*) may begin an object noun phrase, e.g., you, her, some;
(*c*) may continue a noun phrase, e.g., own, last, open-class items;
(*d*) are followed by singular verbs, e.g., this, he, someone;
(*e*) are followed by plural verbs, e.g., these, you, open-class items ending in *-s*;
(*f*) may be followed by either singular or plural verbs, e.g., the, some, her, open-class items ending in *-'s*.

The analysis section of the program has as input simply the code-words associated in the dictionary with the actual words in the input text.

The traditional labels *preposition*, *conjunction* and *adverb* also seem too unspecific. They do not suggest, for example, that words so labelled have at least one function in common. The ambiguity of sentence (2),

(2) The girl guides fish,

disappears in

(12) The girl in the brown hat guides fish,
(13) The girl guides often fish,
(14) The girl who you noticed guides fish.

Prepositions, *conjunctions* and some kinds of *adverb* all reveal, among other things, where a particular noun phrase ends. On the other hand, not all words traditionally labelled as *prepositions*, for instance, share identical functions. If *over* and *at* were simply listed as prepositions, we would not be able to provide two analyses for the ambiguous sentence

(15) He looked over the houses,

but only one for

(16) He looked at the houses.

At is clearly different from *over* in that it must be followed by a noun phrase and cannot be separated from the verb as *over* is in

(17) He looked the houses over.

Different conjunctions also have different functions. Both *whenever* and *before* can introduce whole clauses:

(18) He stopped work whenever he felt tired,
(19) He stopped work before he felt tired.

Before, however, cannot be followed simply by an adjective as *whenever* can in

(20) He stopped work whenever possible,

and neither *before* nor *whenever* can introduce a subject clause, as *how* can in

(21) How he does so much work is a mystery.

The traditional label *adverb* is scarcely more satisfactory. Distinctions between adverbs of time and place do not seem to provide much information about the structure of sentences. Only one form of time adverbial need be specially noted: phrases like *the next day* and *that morning* which can function in the same circumstances as a time adverb like *yesterday*. In the sentence,

(22) The next morning was fine,

the next morning must be analysed as the subject of the sentence. But it is certainly not the subject in

(23) The next morning they parted friends,

and information about such possible time expressions must be entered in the dictionary to permit the correct analysis.

Adverbs of manner, on the other hand, need to be distinguished not so much from adverbs of time or place, as from sentence adverbs and intensifiers. (Intensifiers are words like *very* which can modify adjectives; sentence adverbs are words like *probably* which can modify whole sentences.) The distinction between these three kinds of adverb can be used to resolve ambiguities of the kind found in

(24) He was trying.

Here *trying* may be either an adjective or a verb. There is no ambiguity, however, in these sentences:

(25) He was very trying,
(26) He was strenuously trying.

Intensifiers can only modify adjectives, while adverbs of manner like *strenuously* can only qualify verbs. The two adverbs cannot be exchanged in

(27) He was very happy,
(28) He was strenuously advancing.

However, a sentence adverb like *probably* can occur in either sentence.

(29) He was probably happy.
(30) He was probably advancing.

Sentence adverbs are thus neutral with respect to words ending in *-ing*, and

(31) He was probably trying

is just as ambiguous as sentence (24). Intensifiers and sentence adverbs must be listed in the dictionary. Manner adverbs like *hard* and *fast* must also be listed, but those ending in *-ly* need not be inventoried, as the ending is sufficient to identify them.

SPECIAL CLASSES OF VERB

The dictionary must also contain information about certain classes of verb. The ambiguity in sentence (24) arises partly because the verb *to be* may be followed either by an adjective or by a present participle. A different kind of ambiguity occurs in

(32) They can fish,

where *can* is either an auxiliary or a normal verb meaning *to put in cans.* Auxiliaries like *be*, *can*, *have*, and *ought* must obviously be listed in the dictionary together with information about what endings a main verb that follows them must take.

Auxiliary verbs can reasonably be considered as closed-class items. Other classes of verb, which are not *closed* in quite the same sense, must, for practical reasons, also be listed in the dictionary. Consider the problem posed by the ambiguities in:

(33) He made a good dinner for the cannibal,

(34) He proved a problem.

In these sentences, *make* and *prove* may be acting as normal verbs followed by an object, or as copulative verbs taking a complement, like *be* in

(35) He was a problem.

We could obtain the necessary two analyses for sentences (33) and (34) by including a general rule in the program that any open-class verb may be followed either by an object or by a complement. But then we would also obtain one undesirable analysis for

(36) He solved a problem,

in which *solve* is treated as a potentially copulative verb. One way of preventing such undesirable analyses is to list the small class of possible copulative verbs in the dictionary.

A different but parallel problem is exemplified in this loose paraphrase of sentence (33):

(37) The cannibal found him an excellent meal.

In this sentence, *find* can either be a double-object verb like *give* in

(38) The cannibal gave him an excellent meal,

or an object-complement verb like *consider* in

(39) The cannibal considered him an excellent meal.

Once again it would lead to a number of undesirable analyses if a general rule were included in the program that unlisted verbs may be followed by one object, by two objects, or by an object and a complement. This becomes apparent when we compare

(40) He ate his dog biscuits

with

(41) He gave his dog biscuits.

Information about word-endings reveals only that the phrase *his dog biscuits* contains a possessive article followed by an uninflected open-class item and an open-class item ending in *-s*. It might consist of either one or two noun phrases and the general rule about unlisted verbs would accordingly assign three analyses to each sentence. The number of undesirable analyses is reduced if double-object and object-complement verbs are listed in the dictionary and the general rule for unlisted verbs is that they may be followed by only one object. *Give* and *find*, but not *eat*, then appear as possible double-object verbs. *Find* and *consider*, but not *give* or *eat*, appear as object-complement verbs. The ambiguities of Sentences (37) and (41) are still detected, but sentences (38), (39) and (40) correctly receive only one analysis.

Make has already been noted as a verb which can take either a normal object or a copulative complement. It can also take a double-object as in

(42) She made him a table,

and an object followed by a complement as in

(43) She made him what he is today.

It gives rise to yet another kind of ambiguity in

(44) She made him fish for his supper,

where *make* may be either a double-object verb as in sentence (42) or an object-infinitive verb as in

(45) She made him cry.

Object-infinitive verbs must also be listed in order that they can be distinguished both from other special classes of verb and from normal verbs. Such a distinction enables the program to determine the potential difference in structure between

(46) She lets her house

and

(47) She lets her cry

where the phrases *her house* and *her cry* both have the same superficial appearance of a possessive article followed by an uninflected open-class item.

Two additional classes of special verb might be listed, as they give rise to ambiguity when followed by words ending in *-ing*. This ending has already been noted as characteristic of both verbs and adjectives. It can also act as an ending for words which behave as nominal elements. Thus *reading* is behaving as a nominal element in

(48) Reading is fun,

(49) She writes reading books for children,

but in

(50) Some children avoid reading books

the structure is ambiguous and *reading* may be either a nominal element or a verb. Verbs such as *avoid* and *like*, which permit this kind of ambiguity, can be specially listed. *Like* and several other verbs may also give rise to ambiguity when they are followed by a possessive. In the sentence

(51) I don't like your playing cards,

playing may be either a nominal element or a verb. If ambiguities of this kind are to be detected, and the differences in structure are to be revealed, the most effective procedure seems to be to list in the dictionary information about all the special classes of verb discussed in this paper, and a few other classes not specifically mentioned (see Alexander & Kunz 1964; Pride 1965).

CONCLUSION

This paper has discussed the function of a dictionary in a program for parsing sentences. The program is required to assign at least one satisfactory analysis to every sentence, and more than one analysis where a sentence is, on syntactic grounds, potentially ambiguous. Since the program contains only abstract information about the structure of sentences, it must consult a dictionary to determine the relevant syntactic properties of the actual words in each sentence. The dictionary need only contain information about inflectional word-endings, about exceptions to the routine for determining word-endings, and about words which may give rise to ambiguity or which have otherwise undiscoverable syntactic functions. Such words are listed as special verbs or closed-class items.

The dictionary does not resemble a normal dictionary in either function, content or length. It can be rapidly inspected for each word in any new sentence. Yet the arrangement of information in the dictionary enables the program to handle sentences containing a potentially infinite range of vocabulary.

REFERENCES

Alexander, D., & Kunz, W. J. (1964). Some classes of verbs in English, *Linguistics Research Project mimeo*, Indiana University.

Pride, J. B. (1965). 'Some Classes of English Verbs', *University of Leeds mimeo.*

Thorne, J. P. (1964). Grammar and machines, *Transactions of the Philological Society.*

Thorne, J. P., Dewar, H., Whitfield, H., & Bratley, P (1966). A model for the perception of syntactic structure, *Colloque International sur l'Informatique.* Toulouse.

PROBLEM-ORIENTED LANGUAGES

12

POP-1: AN ON-LINE LANGUAGE

R. J. POPPLESTONE
EXPERIMENTAL PROGRAMMING UNIT
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

POP-1 is an on-line computer language, whereas most of the well-known languages (FORTRAN, MAD, ALGOL, LISP, etc.) are designed for off-line use. POP-1 is for use by a person communicating directly with a computer *via* a typewriter. It differs from JOSS (Shaw 1964) in that it is primarily intended for the sophisticated computer user.* With this in mind I have aimed at a tolerable efficiency of execution, and an ability to define and name new operations, with comprehensive monitoring facilities. On the other hand actual error messages are rather simple.

At the very lowest level, the computer can be used just as a desk calculator. For example

```
23+54+72+98⇒
** 247
```

Lines preceded by double asterisks represent program output.

If the user wants to perform a number of operations using the same number, he can declare a variable to hold it.

```
VARS A;
24·56→A;
97 * A⇒
** 2382·32
(27 * A+3)/(17−5 * A)⇒
** −6·2677
```

* See also MAP (Kaplaw, Strong & Brackett 1966).

All operations are organised by means of a stack, and this basic feature shows up explicitly in the case of two types of instruction:

(i) 'argumentless' output commands (for example ⇒ by itself will cause the number at the top of the stack to be printed);
(ii) 'assignmentless expressions' which are interpreted as implicit assignments to the top of the stack.

With the above examples in mind, certain principles seem obvious.

(i) An on-line language must provide a way of returning the computer to a standard state no matter what state it is in at any time (analogous to the operation of clearing a desk calculator).

(ii) If an error is detected, a message as to the nature of the error must be displayed, and the machine must then enter a clearly defined state, with as little damage as possible to the results already built up.

The on-line user can make best use of such a system by building up complex entities in small units. For example, when calculating a large expression, it is better to work out parts of it and store these parts in variables, rather than try to do the whole thing at once.

In the above examples POP-1 has appeared as a language with a fixed vocabulary. In fact both the vocabulary, and to a limited extent the grammar, are extensible. In the terms of the earlier analogy it is as though we had a calculating machine equipped with an indefinite number of spare keys, on which we can stick new labels. For example, we can define an operation to find the HCF of two numbers by

```
FUNCTION HCF X Y
VARS QUOTIENT REMAINDER
DEFINE X//Y→REMAINDER→QUOTIENT;
IF REMAINDER=0 THEN Y EXIT
HCF(Y, REMAINDER) END
```

Where // produces two values, remainder and quotient. The instinctive reaction of the on-line user, having defined a function, is to test it:

```
HCF(12, 16)⇒
** 4
```

One may decide at this stage that it would be preferable to use an in-fixed operator, say ++ instead of HCF. To do this we need to re-assign the 'value' of the operator HCF, thus:

```
VALUE HCF→VALUE ++;
SETPRECEDENCE("++", 12, "SYMMETRIC");
```

The symbol 'SYMMETRIC' gives the signal that the operator '++' is to be used as an infix.

```
25 ++ 35⇒
** 5
```

The purpose of the SETPRECEDENCE operation is to declare ++ as being an

infixed operator with precedence 12. Thus addition (precedence 6) and multiplication (precedence 5) will be done before ++, i.e.,

```
4+6++7 * 5⇒
**5
```

The line VALUE HCF→VALUE ++ simply assigns to ++ the definition of HCF already established.

POP-1 grammar is a simple precedence grammar. Functions can be written either before their arguments as HCF(X, Y) or after as in X Y HCF.

Functions can be redefined at any time. This is obviously important for an on-line language, since the first definition of a function may be incorrect. In fact, a function definition is no more than an assignment of a constant (the definition) to the name. Thus one could in fact redefine +, *, −, / to have meanings over some field other than the reals (e.g., the complex numbers or a finite field).

For instance to do this for a finite field of characteristic P one writes:

```
VARS P
FUNCTION ADD X Y
DEFINE ADD(X, Y) // P→X→Y; X END
VALUE ADD VALUE+ →VALUE ADD →VALUE+ ;
```

Because of the stack basis of this sequence, the result is to interchange the values of 'ADD' and '+' and similarly for SUB and MULT. Note that the last line has the desired effect of exchanging the values of 'ADD' and '+' because it consists of a sequence of assignments first to, and then from, the stack. Further, since it is executed before the function definition itself is entered (which occurs on the first occasion that the function is called) the operator use of ADD in 'DEFINE ADD (X Y)' already has the meaning previously possessed by +. It is necessary to introduce the names ADD SUB MULT DIV because the old definitions of these operations are needed by the new ones.

Now we get, for example,

```
7→P;
4+5⇒
** 2
```

DIV can be defined using the euclidean algorithm.

Algorithm:

```
FUNCTION DIV X Y DEFINE X * RECIPROCAL (Y) END
FUNCTION RECIPROCAL X
VARS U V H
DEFINE X P EUCLID →U →V →H;

IF NOT (H=1) THEN NL TEXT P NOT PRIME; EXIT
U END
FUNCTION EUCLID X Y
VARS QUOTIENT REMAINDER U V H
DEFINE X//Y→REMAINDER →QUOTIENT',
```

```
IF REMAINDER=0 THEN Y, 1, 0 EXIT
Y REMAINDER EUCLID →U →V →H;
H, U SUB(U, MULT (V, QUOTIENT)), V END
```

The symbol NL means 'new line', i.e., it is a formal function.

The effect of VALUE is to make the next word be treated as a variable. Its opposite is OBEY which makes the next word be treated as a function.

ARRAYS

These are treated as functions with a difference—the difference being that they have a meaning when preceded by an → (apart from the above meaning). Declaring that a variable is to hold an array is distinct from assigning the space to hold that array. The word ARRAY in any list of variables indicates that all following variables are to be treated as arrays (until NORMAL or FUNCTION or the end of the list is reached). Rectangular arrays are created by the function NEWARRAY. Thus, to handle permutations, one might write the following group of functions:

```
VARS LENGTH;
FUNCTION READPERM
VARS ARRAY X NORMAL I
DEFINE 1→I; ⟨⟨1 LENGTH⟩⟩NEWARRAY→ VALUE X;
L 1: NEXTATOM → X(I);
IF I=LENGTH THEN VALUE X EXIT
I+1→I; GOTO NEXT END
```

The brackets ⟨⟨ ⟩⟩ enclose items which are to be evaluated and then made into a list

```
FUNCTION MULTPERM ARRAY X Y
VARS I ARRAY Z
DEFINE 1→I; 1 LENGTH NEWARRAY→VALUE Z;
NEXT: I X Y→ Z(I);
IF I=LENGTH THEN VALUE Z EXIT
I+1→I; GOTO NEXT END
3→LENGTH;
VARS A B C D
READPERM 1 2 3→A; READPERM 3 2 1→B; READPERM 1 3 2→C;
VALUE MULTPERM → VALUE *;
A * B⇒
** 3 2 1
B * C⇒
** 2 3 1
```

Since arrays are created by function, one can write functions to create special arrays, e.g., in a chess program

```
FUNCTION NEWBOARD
VARS I J K ARRAY A
```

```
DEFINE <<1 8 1>> NEWARRAY→VALUE A;
1→I;
L1: IF I>8 THEN VALUE A EXIT
1→J;
L2: IF J>8 THEN I+1→I GOTO L1
ALSO (I MASK 1)+(J MASK 1) MASK 1 THEN "BLACK" ELSE "WHITE"
ALSO →A(I, J), J+1→J GOTO L2 END.
```

Quotation marks enclose atoms. MASK forms the logical AND of its arguments considered as bit patterns. When the conditional THEN ··· ELSE ··· ALSO is obeyed, the commands between THEN and ELSE are obeyed if the top of the stack is 'true' (i.e., non zero) before THEN. Otherwise the commands between ELSE and ALSO are obeyed. ALSO is thus used as a terminator of conditionals rather similar in action to the semi-colon which is used to terminate assignments.

LIST PROCESSING

POP-1 was originally a pure list-processing language, and list-processing facilities are well developed. Let us consider the function REV to reverse a list.

```
FUNCTION REV X
DEFINE IF X ATOM THEN X EXIT
REV(X TL) <> X HD :: 1 END
```

The operator <> means join, i.e., concatenate the lists which are its operands, and :: is CONS in LISP.

Let us translate some dog-English into dog-French. Suppose we have read an English sentence as a list, and suppose we have a dictionary DIC giving what part of speech each English word is, and its French equivalent, together with information as to whether a word (if it is an adjective) comes before or after the word it qualifies. If a sentence is defined as a nounphrase followed by a transitive verb followed by a nounphrase then:

```
FUNCTION SENTENCE X D
VARS U V T
DEFINE NOUNPHRASE (X, D)→T→V;
  "SENTENCE" :: V <> LOOKUP (T HD, D) HD <> (T TL NOUNPHRASE→U)::1;
  U END
FUNCTION NOUNPHRASE X D
  VARS U V T
  DEFINE LOOKUP(X HD, D)→U;
  IF U HD="NOUN" THEN "NOUNPHRASE" : U :: 1; X TL EXIT
  NOUNPHRASE(X TL, D)→T→V;
  V <> U :: 1; T END
```

These functions work by finding the first syntactic unit of the type that bears their name in the list X, giving as a result an analysis of that unit, and whatever remains in the list X. Having produced a parsed version of the sentence, we translate it with the following function:

```
FUNCTION TRANSLATE X G
DEFINE
  IF X HD="SENTENCE" THEN TRANSLATE (X TL HD, G); FRENCH (X TL TL
    HD) SPR;
  TRANSLATE (X TL TL TL HD, G) EXIT
  IF X HD="NOUN" THEN FRENCH (X) SPR EXIT
  IF X HD="ADJECTIVE" THEN IF G="MASCULINE" THEN FRENCH (X) HD SPR
    EXIT
  FRENCH (X) TL HD SPR EXIT

  IF X HD="NOUNPHRASE" THEN GENDER (X TL HD)→G;
  TRANSSELECT (X TL YL, G, "ARTICLE");
  TRANSSELECT (X TL, TL, G, "BEFORE");
  TRANSLATE (X TL HD, G);
  TRANSSELECT (X TL TL, G, "AFTER")
  EXIT
  NL "FAIL" PR  END
```

This works as follows. X is the tree representing the sentence, or a part of it, in the parsed form, containing all the information necessary for a translation. G is the gender of the part of the sentence being translated, if this is relevant. Thus the sentence 'The cat eats fish' is parsed as

```
SENTENCE

  NOUNPHRASE                VERB      NOUNPHRASE

    NOUN      ADJECTIVE     MANGE      NOUN      ADJECTIVE

      CHAT MASC [LE LA LES LES] ARTICLE  POISSON MASC

                                        [LE LA LES LES] ART
```

Note that lists are enclosed in square brackets.

When applying TRANSLATE to this structure, first we apply TRANSLATE to the nounphrase (X TL HD), then we find the French equivalent of the verb, and print it preceded by a space. To translate the nounphrase, we first extract the gender of the noun (GENDER (X TL HD)). We then apply the function TRANSSELECT to the list of adjectives following the noun.

TRANSSELECT is defined by:

```
FUNCTION TRANSSELECT X G S
DEFINE
NEXT: IF X ATOM THEN EXIT
IF POSSELECT (X HD)=S THEN TRANSLATE (X HD, G) ALSO
X TL → X; GOTO NEXT END
```

Thus transselect translates only those items in the list X which have the atom S in the position POSSELECT. Hence the first call translates the articles 'the' and 'a', and the second those adjectives that come before the noun in French. Next, we translate the noun itself, and then those adjectives which come after the noun.

For curiosity's sake, Fig. 1 shows some worked examples done on the console typewriter as a demonstration.

MACROS

Since POP-1 is a list-processing language, one might expect that it would be easy to provide facilities for writing macros. This is indeed the case. First we define the function MACRO by

```
FUNCTION MACRO          VARS X
DEFINE READ→X;
  SETTYPE(X,5,3,15,0);
  <<"FUNCTION" X>> <> INSTREAM→INSTREAM
END
SETTYPE(MACRO,5,3,15,0);
```

This is in fact itself a macro definition, and works as follows. The final SETTYPE command has the effect of making the function MACRO be executed as soon as it is read. Thus if

```
MACRO THIRD . . .
```

occurs in the input instream, then when MACRO is read, it is immediately obeyed. Thus the first action of MACRO is to read the next word, i.e., THIRD off the input stream and store it in X. INSTREAM is a list which, if non-empty, contains the atoms which are next to be read off the input stream. Thus after the evaluation of MACRO the next items to be read off the input stream will be

```
FUNCTION THIRD . . .
```

Thus the effect of MACRO . . . is to define THIRD as a function which is itself evaluated as soon as it is read. Now let us consider how to define THIRD as a macro which will take the third member of a list.

```
  MACRO THIRD
VARS X
  DEFINE MACARGS→X;
    <<"(""("X>> <>[]TL TL HD)] <> INSTREAM→ INSTREAM END
```

Let us see that THIRD defined above will produce the correct reverse Polish input for the assembler.

Suppose THIRD is called by

```
THIRD (A <> THIRD(B))
```

as soon as THIRD is read (and its precedence has been set by settype so that it is not transferred behind the list of arguments, which would be its fate if it were an ordinary function) it is evaluated. The function MACARGS, which is

```
23 + 54 + 72*96 - 298*34 =>

** -3143


VARS A B C%

[Semicolon  Does not occur on the typewriter so use % instead]

19 -> A%  25 -> B%
A*B + 98*(75 + B) + 4*A =>

** 10351

FUNCTION REV X
DEFINE
    IF X ATOM THEN NIL EXIT
    REV(X TL) <> X HD END

REV( [1 2 3 4 5 6]) =>

** 1
[No good so redefine REV]

FUNCTION REV X
DEFINE
    IF X ATOM THEN NIL EXIT
    REV(X TL)  <>  <<X HD >> END


REV( [1 2 3 4 5 6] ) =>

** [6 5 4 3 2 1 ]

[OK. Now load translation program from paper tape]


[THE BIG BLACK CAT EATS THE RED MEAT] SENTENCE

**PLEASE TELL ME ABOUT "BIG"
**PART OF SPEECH                                ADJECTIVE
**FRENCH
**SINGULAR                                      GRAND
**FEMININE                                      GRANDE
**PLURAL                                        GRANDS
**FEMININE                                      GRANDES
**
**IS IT BEFORE OR AFTER OR ARTICLE              BEFORE
**PLEASE TELL ME ABOUT "CAT"
**PART OF SPEECH                                NOUN
**FRENCH                                        CHAT
**IS IT MASCULINE OR FEMININE                   MASCLINE
**YOU MUST REPLY ONE OF MASCULINE OR FEMININE
                                                MASCULINE
```

FIG. 1. An example of a POP-1 "session".

```
**PLEASE TELL ME ABOUT "RED"
**PART OF SPEECH                          AJECTIVE
**I DO NOT KNOW THIS PART OF SPEECH
**PART OF SPEECH                          ADJECTIVE
**FRENCH
**SINGULAR                                ROUGE
**FEMININE                                ROUGE
**PLURAL                                  ROUE##ROUGES
**FEMININE                                ROUGES
**
**IS IT BEFORE OR AFTER OR ARTICLE        AFTER

-> A -> B%

A =>
** 1   [This is the null list, and is the unparsed part of
the input]

B =>
** [SENTENCE [NOUNPHRASE [NOUN CHAT MASCULINE ]
**[ADJECTIVE [NOIR NOIRE NOIRS NOIRES ]AFTER ]
**[ADJECTIVE [GRAND GRANDE GRANDS GRANDES ]BEFORE ]
**[ADJECTIVE [LE LA LES LES ]ARTICLE ]][VERB MANGE
**][NOUNPHRASE [NOUN VIANDE FEMININE ][ADJECTIVE
**[ROUGE ROUGE ROUGES ROUGES ]AFTER ][ADJECTIVE
**[LE LA LES LES ]ARTICLE ]]]

[This is the tree structure of the sentence represented in
bracketed form - Let us explore this tree]

3 HD =>
** SENTENCE

B TL HD =>
** [NOUNPHRASE [NOUN CHAT MASCULINE ][ADJECTIVE
**[NOIR NOIRE NOIRS NOIRES ]AFTER ][ADJECTIVE
**[GRAND GRANDE GRANDS GRANDES ]BEFORE ][ADJECTIVE
**[LE LA LES LES ]ARTICLE ]]

B TL TL HD =>
** [VERB MANGE ]

B TL TL TL HD =>
** [NOUNPHRASE [NOUN VIANDE FEMININE ][ADJECTIVE
**[ROUGE ROUGE ROUGES ROUGES ]AFTER ][ADJECTIVE
**[LE LA LES LES ]ARTICLE ]]

B TL TL TL TL HD
ERROR 30 COMPILER   HD
ERROR 30
[An attempt has been made to evaluate HD with an atomic argument]

B 1 TRANSLATE
 LE GRAND CHAT NOIR MANGE LA VIANDE RO
UGE

FUNCTION SEN##TRANS X
DEFINE X SENTENCE TRANSLATE END

[THE LITTLE B##BLACK BOY EATS THE MEAT] TRANS
 LE PETIT GARCON NOIR MANGE LA VIANDE

[ The character  "##"  causes the atom just typed to be ignored]
```

a library function defined in POP-1, reads the string of atoms (A <> THIRD (B)) and converts it into the list [A <> THIRD(B)] (square brackets denote list structures, round brackets are just symbols). Assuming INSTREAM to be empty (i.e., we are not in the middle of a macro call), this first call of THIRD will leave

((A <> THIRD(B)) TL TL HD)

in INSTREAM. Processing this will result in A being compiled and <> being set aside before THIRD is called again. Thus at the second call of THIRD INSTREAM will contain (B)) TL TL TL HD), and the second call will turn this into ((B) TL TL HD)) TL TL HD). Hence B TL TL HD will now be compiled, and then the <> operator which was set aside, and then the TL TL HD, so the total effect of the macro calls is

A B TL TL HD <> TL TL HD

which is the correct reverse Polish form of the input stream.

IMPLEMENTATION

The present implementation (on an Elliott 4100) is by means of an interpreter. A program being input is first processed to set variables and functions into the correct reverse Polish order. The resulting stream is then translated into pseudo-machine code which is obeyed by the interpretive routine. Certain words such as FUNCTION, THEN, EXIT have the effect of activating special routines as soon as they are input. The user himself can define such words, e.g., in making macro definitions. Thus the whole system is easily extensible.

AIMS

POP-1 was originally conceived as a list-processing language which has acquired an increasingly on-line emphasis. It is now adopted as a language for exploratory programming projects at the Experimental Programming Unit of the University of Edinburgh, and as such will be incorporated in the on-line time-sharing system being developed there.

EDITOR'S NOTE

POP-1 has now been superseded by a greatly extended version, POP-2, which is the subject of Chapter 14.

REFERENCES

Kaplaw, R., Strong, S., & Brackett, J. (1966). MAP: a system of on-line mathematical analysis. *Technical report MAC-TR-24*, Massachusetts Institute of Technology.

Shaw, J. C. (1964). JOSS: A designer's view of an experimental on-line computing system. In *AFIPS Conference Proceedings*, Vol. 26. Baltimore and London: Spartan Books.

13

SELF-IMPROVEMENT IN QUERY LANGUAGES

J. M. FOSTER
MINISTRY OF AVIATION
ROYAL RADAR ESTABLISHMENT
GREAT MALVERN, WORCS.

INTRODUCTION

A program, by means of which untrained operators could question a computer on-line about a store of data, was demonstrated in an elementary form by the author (Foster 1967). A user can question the computer in a subset of English and the computer can ask questions back. The language which is to be used is specified to the program initially, by means of a set of syntactic rules and associated λ-calculus expressions. A feature of the program is its ability to accept definitions of new phrases and words in terms of existing ones. For example, the user can say

> The average x of y, where x is a noun and y is a noun-phrase, is a noun-phrase and means 'the sum of the age of each of y divided by the number of y'.

The computer then generates a new syntactic rule and λ-expression in such a way that if a particular example is presented, say

> the average age of the men

then it is exactly as if

> the sum of the age of each of the men divided by the number of the men

had been the phrase used.

The demonstrations, though indicating that the method was workable, left unanswered the question whether, on the scale of a practical example, the time and space used would be tolerable. A particular doubt concerns the

definition of new phrases by the users. If the definitions, each of which was implemented by adding a new syntactic rule to the system, were allowed to proliferate then the analysis stage of the program would almost certainly become too slow. The problem which this paper discusses is that of reorganising the syntactic definitions and their λ-expressions.

This is an example of a very general and difficult problem, that of learning to recognise data with structure. Instead of tackling this problem generally the paper presents some assumptions about structures and some rules for organising them which are probably valid in the field in question.

We wish to have a program which will be given pairs of strings of characters, which are assumed to have structures in some correspondence, representing the sentence and its translation into λ-expressions. The program will organise a description of these so that later, when it is presented with a sentence, it will associate a translation with it. The program should satisfy the following restrictions.

1. If presented with one of the original examples, it should associate with it the string which was given as its pair.

2. If it recognises any strings which are not the original strings, then the translation it associates with them should be sensible.

3. If it makes assumptions which turn out to be wrong it should be capable of reorganising the description to remove the fault.

For example, if it is presented with the following pairs

a	Pick, a
b	Pick, b
$a+b$	Pick, a; Add, b
c	Pick, c

it might perhaps make the following translation,

$c+a$	Pick, c; Add, a
$a+b+c$	Pick, a; Add, b; Add, c

The next section presents the assumptions that are made, the section following it gives some rules by means of which the data can be arranged and the penultimate section discusses the application of the rules to some examples of typical query language situations.

A program to apply the rules is currently partly working.

It is not suggested that the program is directly applicable in other fields, but some related pairs of structural strings in other problems are:

1. The set of operations carried out by the λ-expression evaluator and the results produced. Improvement here would enable often-applied groups of functions to be treated as special cases (e.g., whether a function should be computed or kept as a table).

2. Sentences and actions (indeed the general problem of learning a language).

3. Physical stimuli and reactions (e.g., balancing).

4. A picture and its description (in words or actions).

DISCUSSION OF ASSUMPTIONS

The program tries to form a syntactic description which generates translation strings from input strings in the following way. Suppose that one of the substitution rules of the grammar is

$$S \rightarrow A\ B\ C$$

where S is a class-name and A, B and C are either class-names or basic symbols (words). Each such rule has with it a string of characters

$$S \rightarrow A\ B\ C \qquad pq3r21s1$$

in which the small letters represent constants and the numbers represent the outputs of the first, second, etc., parts of the rule. This string is the output of this rule. For example, the grammar

$S \rightarrow P$	Pick, 1
$S + P$	1; Add, 3
$P \rightarrow a$	a
b	b
c	c

would perform the translations at the end of the introduction. We are trying, therefore, to produce a grammar like this from a set of example pairs. Since the query language program contains its translations in the form of λ-expressions, these are used for the rule meanings. However, this is a very simple use of λ-expressions.

A program which recognises similarities between structures and has some ability to generalise must operate against an opposing force lest it decide that everything is like everything else. In this case the opposing force is provided by the requirement that the translation string should correspond correctly with the example string. Since the process of translation is controlled by the syntax rules, this requirement considerably restricts the alterations which can be made to the syntax, in a way which seems both sensible and effective.

Accordingly, no transformations are made unless the following condition is satisfied. It amounts to ensuring that the translation process described is adequate. Suppose that a rule is

$$S \rightarrow A\ B\ C\ D \qquad p12q43r2q43$$

and that parts of the rule are being considered to see whether they can be made into rules. Then it would be permissible to write

$S \rightarrow A\ B\ X$	$p12q3r2q3$
$X \rightarrow C\ D$	21

or

$S \rightarrow A\ B\ X$	$p123r23$
$X \rightarrow C\ D$	$q21$

or

$S \rightarrow A\ X$	$p12r2$
$X \rightarrow B\ C\ D$	$1q32$

but the attempt to write, say,

$$S \rightarrow A\ X\ D$$
$$X \rightarrow B\ C$$

would fail because translation strings could not be generated. Hence, if we are trying to make a new rule from a part of an old one, three conditions must be satisfied.

1. A unique piece of the translation string must correspond to the part; though this identical piece may occur several times.

2. Nothing corresponding to the part may occur outside the corresponding pieces of the translation string.

3. Since the translation strings are λ-expressions in the query language application, the new rule must be a complete expression.

This condition can be expressed by saying that if parts of sentences are grouped in the syntax then they will also be grouped in the translation. This assumption is a reasonable one for simple English sentences. No harm is done to the system by a counter-example, which will simply be learnt at a higher level in the syntax, that is, as an idiom.

The other assumption is that words and phrases used in similar contexts and with similar meanings (in the sense of the above constraint) can usefully be combined in a syntactic class. So

$S \rightarrow A\ B\ C$	$3rp21$
$D\ E\ C$	$3rq12$

could give

$S \rightarrow X\ C$	$2r1$
$X \rightarrow A\ B$	$p21$
$D\ E$	$q12$

This example introduces no new possible input strings but we might say that

$S \rightarrow A\ B\ C$	123
$D\ B\ E$	123

gives

$S \rightarrow X\ B\ Y$	123
$X \rightarrow A$	1
D	1
$Y \rightarrow C$	1
E	1

which introduces as possible examples of S, $A\ B\ E$ and $D\ B\ C$. Clearly this can introduce errors, but this will do no harm if an unlearning mechanism is present to change the rules back if necessary.

The next section gives the details of the algorithms which are being used to implement this assumption.

TRANSFORMATION ALGORITHMS

The examples of the section following may shed light on the reasons for these rules.

Learning Rule 1

Consider all the syntactic definitions in pairs. If there are pieces which are common to two definitions which satisfy the constraint of the last section, then they may be made into a new rule. This is done unless the piece consists of a single class name.

$A \rightarrow B\ C\ D$	123
$E \rightarrow G\ B\ C\ F$	1423

gives

$A \rightarrow X\ D$	12
$E \rightarrow G\ X\ F$	132
$X \rightarrow B\ C$	12

The application of this rule does not change the sentences that are recognised by the program, and hence there is no need to undo this process. It merely gives a more condensed description of the sentences.

Learning Rule 2

If

$$\begin{array}{ll} Q \rightarrow T & 1 \\ \quad A\ B\ C & \\ T \rightarrow D\ E & \\ \quad F\ G & \end{array}$$

where $A\ B\ C$, $D\ E$, etc., are any sets of symbols, then replace Q by T and add $A\ B\ C$ to the alternatives for I.

This rule covers the case when things are recognised to be similar in different places. For example, if

$$\begin{array}{l} Q \rightarrow A\ B \\ \quad C\ D \\ T \rightarrow A\ B \\ \quad C\ D \end{array}$$

and the translations are identical, then the following sequence guesses that Q and T are identical.

$$\begin{array}{l} Q \rightarrow X \\ \quad Y \\ T \rightarrow X \\ \quad Y \\ X \rightarrow A\ B \\ Y \rightarrow C\ D \end{array}$$

(by Learning Rule 1)

$$\begin{array}{l} T \rightarrow X \\ \quad Y \\ X \rightarrow A\ B \\ \quad Y \\ Y \rightarrow C\ D \end{array}$$

and Q is rewritten as X (LR2 on Q)

$$X \rightarrow A\ B$$
$$\quad\ Y$$
$$Y \rightarrow C\ D$$

and T is rewritten as X (LR2 on T)

Finally

$$Y \rightarrow A\ B$$
$$\quad\ C\ D$$

and X is rewritten as Y (LR2 on X)

This rule can introduce errors, so an inverse is needed for when they are detected.

Unlearning Rule 2

$$Q \rightarrow A\ C$$
$$\quad\ B\ D$$
$$R \rightarrow \ldots Q \ldots$$

and $Q \leftrightarrow A\ C$ in R

gives $Z \leftrightarrow A\ C$

$$Q \rightarrow A\ C$$
$$\quad\ B\ D$$
$$Z \rightarrow B\ D$$
$$R \rightarrow \ldots Z \ldots$$

where the significance of $\leftrightarrow$ is that this rule must not be generated by any transformations.

Learning Rule 3

For all the definitions of a single class-name, pick out the similar pieces satisfying the constraint and examine the remaining corresponding pieces. The largest corresponding pieces which satisfy the constraint are formed into a rule. For example, if

$A \rightarrow B\ C\ D\ E\ F$	21543
$\quad H\ D\ J\ K$	1342

gives

$A \rightarrow X\ D\ Y$	132
$X \rightarrow B\ C$	21
$\quad H$	1
$Y \rightarrow E\ F$	21
$\quad J\ K$	12

This rule again can introduce errors, since it implies that

$A \rightarrow H\ D\ E\ F$	1432
$\quad B\ C\ D\ J\ K$	21453

which might or might not be the case. We therefore need an unlearning rule.

Unlearning Rule 3

$$P \rightarrow X\ C\ Y$$
$$X \rightarrow A\ B$$
$$R\ S$$
$$Y \rightarrow D\ E$$
$$T\ U$$

and $P \leftrightarrow A\ B\ C\ T\ U$

gives

$$P \rightarrow A\ B\ C\ Z$$
$$D\ E\ C\ Y$$
$$Y \rightarrow D\ E$$
$$T\ U$$
$$Z \rightarrow D\ E$$
$$Z \rightarrow T\ U$$

All these rules are based on the assumption of section 2 and satisfy the constraint of that section.

EXAMPLES

1. We consider an example showing how the word average, used in a context like the word sum, is assumed to operate in a different context by analogy with the properties of 'sum'.

Suppose that part of a grammar is

np→*np* divided by *np*	div(1, 4)
np of *np* of all *np*	1(3, 6)
the noun	2
the noun of *np*	2(4)
np of *np* and *np*	1(value, set(3, 5))
noun→sum	sigma
number	count
ages	age
men	men

The translation of 'the sum of the ages of the men'

is

sigma (age, men)

where the definition of sigma is the sum of the value of the first parameter applied to each member of the class which is the second parameter.

The translation of 'the sum of *a* and *b*'

is

sigma (value, set(*a*, *b*))

where set makes its parameters into a single class and value gives the numerical value of a class member.

If in this grammar we define

'the average of x of all y' where x and y are *np*, is a *np* and means 'the sum of x of all y divided by the number of y'

then we obtain (1) a new rule for *np*

$$np \rightarrow \text{the average of } np \text{ of all } np$$

$$(\lambda xy.\,\text{div (sigma } (x, y), \text{count}(y)))(3, 6)$$

LR3 applied to this definition and the second of *np* gives

$np \rightarrow X$ of *np* of all *np*	1(3, 6)
$X \rightarrow np$	1
the average	$(\lambda xy.\,\text{div (sigma } (x, y), \text{count } (y)))$

LR2 then gives

$np \rightarrow np$ of *np* of all *np*	1(3, 6)
the average	$(\lambda xy.\,\text{div (sigma } (x, y), \text{count}(y))$

LR3 applied to the latter rule and the third of *np* gives

$np \rightarrow$ the Y	2
$Y \rightarrow$ noun	1
average	$(\lambda xy.\,\text{div (sigma } (x, y), \text{count}(y)))$

and LR2 then gives

$np \rightarrow$ the noun	2
noun $\rightarrow$ average	$(\lambda xy.\,\text{div (sigma } (x, y), \text{count}(y)))$

If to this new grammar the phrase 'the average of a and b' is presented, it translates it as

$$(\lambda xy.\,\text{div (sigma } (x, y), \text{count } (y))) \text{ (value, set } (a, b))$$

which happens to be correct.

2. If the definition in the last example had been

'the average of the ages of all the men' is a *np* and means 'the sum of the ages of all the men divided by the number of the men'

then the final result would have been the same, since 'the ages' and 'the men' would have been taken to be *np* by application of LR1.

3. If the program can recognise word suffixes like -s or -ed then it will assume that if two words are used similarly and one forms a plural with -s then so does the other. It can then unlearn this if it finds itself wrong by using UR3.

4. This sequence of arithmetic expressions will produce the following grammar

a	Pick, a
b	Pick, b
c	Pick, c
$a+b$	Pick, a; Add, b
$c+b$	Pick, c; Add, b
$a+c$	Pick, a; Add, c

$b+c$	Pick, b; Add, c
$a+b+c$	Pick, a; Add, b; Add, c
$S\rightarrow Y\ X$	Pick, 12
$Y\rightarrow a$	a
b	b
c	c
$X\rightarrow\phi$	
$+\ Y\ Z$	; Add, 23
$Z\rightarrow\phi$	
$+\ Y$	; Add, 2

where ϕ is the empty rule.

Note that this grammar only recognises three term sums. That is, the similarity between a, b and c has been noticed, but not the possibility of producing the grammar of the second section.

CONCLUSIONS

A program which uses the correspondence between an input string and its translation is being written to recognise some structural properties of the strings. A certain amount of generalisation (and corresponding unlearning) have been built in. The rules used can be justified only in so far as they work.

REFERENCE

Foster, J. M. (1967). Interrogation Languages, *Machine Intelligence 1*, 267-276, ed. Collins & Michie. Edinburgh: Oliver and Boyd.

POP-2 REFERENCE MANUAL

BY
R. M. BURSTALL
AND
R. J. POPPLESTONE
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

CONTENTS

1. INTRODUCTION

1.1. Aims

The following are the main design objectives for the POP-2 language:

(*i*) The language should allow convenient manipulation of a variety of data structures and give powerful facilities for defining new functions over them.

(*ii*) The language should be suitable for taking advantage of on-line use at a console, i.e. it should allow immediate execution of statements and should have a sufficiently simple syntax to avoid frequent typing errors.

(*iii*) A compiler and operating system should be easy to write and should not occupy much storage.

(*iv*) The elementary features of the language should be easy to learn and use.

(*v*) The language should be sufficiently self-consistent and economical in structure to allow it to incorporate new facilities when extensions are desired.

In attaining these objectives certain other desirable features of programming languages had to be relegated to secondary importance:

(*vi*) Fast arithmetical facilities on integer and real numbers.

(*vii*) Fast subscripting of arrays.

(*viii*) A wide variety of elegant syntactic forms.

Naturally whether (*iii*) or (*vi*) and (*vii*) are attained is to a considerable extent a matter of implementation.

1.2. Main features

The following main features are provided. Roughly analogous features of some other programming languages are mentioned in brackets as a guide:

(*i*) Variables (cf. ALGOL but no types at compile time).

(*ii*) Constants (cf. ALGOL numeric and string constants, LISP atoms and list constants).

(*iii*) Expressions and statements (cf. ALGOL).

(*iv*) Assignment (cf. ALGOL, also CPL left-hand functions).

(*v*) Conditionals, jumps and labels (cf. ALGOL but restrictions on jumps and labels).

(*vi*) Functions (cf. ALGOL procedures but no call by name, cf. CPL and ISWIM for full manipulation of functions).

(*vii*) Arrays (cf. ALGOL; cf. CPL for full manipulation of arrays).

(*viii*) Records (cf. COBOL, PL/1, Wirth-Hoare ALGOL records, CPL nodes).

(*ix*) Words (cf. LISP atoms).

(*x*) Lists (cf. LISP, IPL-V).

(*xi*) Macros.

(*xii*) Use of compiler during running (cf. LISP, TRAC, FORMULA ALGOL).

(*xiii*) Immediate execution (cf. JOSS, TRAC).

Notes:

LISP: LISP 1.5

CPL: See Barron, D. W., *et al.* 1964. The main features of CPL, *Computer J.*, **6**, 134-43.

CPL reference manual. Edited C. Strachey (privately circulated).

Wirth-Hoare ALGOL: See Wirth, N., and Hoare, C. A. R. 1966. A contribution to the development of Algol, *Communs Assn Comput. Mach.*, **9**, 413-32.

TRAC: See Mooers, C. N. 1966. TRAC, a procedure describing language for the reactive typewriter, *Communs Assn Comput. Mach.*, **9**, 215-24.

ISWIM: See Landin, P. J. 1966. The next 700 programming languages, *Communs Assn Comput. Mach.*, **9**, 157-166.

1.3. Examples

The following is an example of POP-2 program text. The sign ⇒ (not to be confused with that used in section 1.5 'Notation for functions') prints out some results on a newline prefixed with two asterisks. These results are included in the text below, as they would appear if the program were run on-line at a console.

```
comment arithmetic;
12·0+2·5*(1.5+2.5)⇒
**22.0
vars a b sum;
2*2→a; 3*a→b; a*a+b*b→sum; sum ⇒
**160
function sumsq x y;
    x*x+y*y
end;
sumsq(a, b)+1⇒
** 161
```

```
function fact n; vars p;
    1→p;
loop: if n=0 then p else n*p→p; n-1→n; goto loop close
end;
fact(fact(3))⇒
** 720

comment arrays;
vars a i j;
10→i; 20→j;
newarray([%1, i, 1, j%], sumsq)→a;
a(2, 3)⇒
** 13
10→a(2, 3); a(2, 3)⇒
** 10
function arraysum a1 a2 m n;
    newarray ([%1, m, 1, n%], lambda i j; a1(i, j)+a2(i, j) end)
end;
arraysum (a, a, 10, 20)→a; a(2, 3)⇒
** 20
comment lists;
vars u;
1→i; 2→j;
[%i, i+j, "dog", "cat" %]→u; u⇒
** [1 3 dog cat]
cons ("pig", u)⇒
** [pig 1 3 dog cat]
function append x y;
    if null (y) then [% x %] else cons(hd(y), append(x, tl(y))) close
end;
append(4, [% 1, i+1, 3%])⇒
** [1 2 3 4]

comment records;
vars consper destper forename surname male p1 p2;
recordfns("person", 500, [0 0 1])→male→surname→forename
                                →destper→consper;
consper("jane", "jones", false)→p1; consper("sam", "smith", true)→p2;
surname(p1)⇒
** jones
datalist(p1)⇒
** [jane jones 0]
routine marry x y;
    if male(x) and not(male(y)) then surname(x)→surname(y) close
```

```
end;
marry(p2, p1); datalist(p1)⇒
** [jane smith 0]
```

1.4. Notation for syntactic description

We use the BNF (Backus-Naur Form) notation as used in the ALGOL report:

::= indicates a syntax definition;
⟨ ⟩ are used to enclose the name of a syntax class;
| denotes disjunction (union of syntax classes).

Concatenation denotes concatenation of any elements of two syntax classes.

We also use a convenient extension of this notation due to R. A. Brooker:

* means that a class may occur n times, $n \geqslant 1$;
? means that a class may occur n times, $n=0$ or 1;
*? means that a class may occur n times, $n \geqslant 0$,

e.g. the definitions

⟨*astring*⟩::=⟨*a*⟩ ⟨*astring*⟩ | ⟨*a*⟩
⟨*bstring*⟩::=⟨*b*⟩ ⟨*astring*⟩
⟨*cstring*⟩::=⟨*c*⟩ ⟨*astring*⟩ | ⟨*c*⟩

may be replaced by

⟨*bstring*⟩::=⟨*b*⟩ ⟨*a**⟩
⟨*cstring*⟩::=⟨*c*⟩ ⟨*a**?⟩

The characters ⟨, ⟩ and * are used in the POP-2 reference language but no confusion should arise.

When we wish to give examples of a syntax class we use the symbol 'e.g.::=', for example:

⟨*bstring*⟩*e.g.*::=⟨*b*⟩ ⟨*a*⟩ | ⟨*b*⟩ ⟨*a*⟩ ⟨*a*⟩ ⟨*a*⟩

The character set of the POP-2 reference language is as follows.

⟨*letter*⟩::=*a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*|*n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*
⟨*digit*⟩::=0|1|2|3|4|5|6|7|8|9
⟨*sign*⟩::=+|−|*|/|$|&|=|⟨|⟩|:|£|↑
⟨*separator*⟩::=,|;
⟨*period*⟩::=.
⟨*sub ten*⟩::=$_{10}$
⟨*bracket*⟩::=(|)|[|]
⟨*bracket decorator*⟩::=%
⟨*quote*⟩::="
⟨*string quote*⟩::='|`

Letters may be written in lower case, upper case or **heavy type** without any change of meaning. It will be conventional however to use **heavy type** letters for syntax words, i.e. those identifiers such as **function, then, end** and **cancel** which have a special meaning for the POP-2 compiler and which characterise certain syntactic forms.

Spaces, tabulate and new lines terminate identifiers, integers, reals and words but otherwise they are ignored.

A distinction is made between the reference language used in this document and a number of possible hardware languages used by particular computer implementations of POP-2. Each character in the reference language should be represented by a distinct character or sequence of characters in the hardware language. A particular letter, whether upper case, lower case, heavy type or not is regarded as the same character in the reference language.

The symbols → and ⇒ used in this paper should be read as a typographical abbreviation for the pairs of characters $-$ > (minus greater than) and = > (equals greater than) respectively.

1.5. Notation for functions

It is convenient to have a notation to specify the domain and range of functions. We will consider functions having several arguments (or possibly none) and producing several results (or possibly none), the notion of functions with more than one result being an extension of normal mathematical usage (*see* section 4.2 'Application of functions'). We introduce a special symbol '⇒' which is not to be confused with any identifier in the POP-2 language.

Suppose $d1, d2, \ldots, dm$ and $r1, r2, \ldots, rn$ are all sets of items. Then $d1, d2, \ldots, dm \Rightarrow r1, r2, \ldots, rn$ is the set of all functions whose domain is $d1, d2, \ldots, dm$ and range $r1, r2, \ldots, rn$, i.e. with arguments which are m-tuples in $d1 \times d2 \times \ldots \times dm$ and with results which are n-tuples in $r1 \times r2 \ldots \times rn$. We express the fact that a function f is a member of this set of functions by

$$f \in d1, d2, \ldots, dm \Rightarrow r1, r2, \ldots, rn$$

Some examples will make this clear.

add ∈ *integer, integer* ⇒ *integer*
divrem ∈ *integer, integer* ⇒ *integer, integer*

where *divrem* is 'divide with remainder', e.g. *divrem* (7, 3) = 2, 1 and *divrem* (14, 4) = 3, 2

roundup ∈ *real* ⇒ *integer*
prime ∈ *integer* ⇒ *truthvalue*

If the function has no results we use an empty pair of parentheses, thus:

printout ∈ *integer* ⇒ ()

The arguments or results may themselves be functions

differentiate ∈ (*real* ⇒ *real*) ⇒ (*real* ⇒ *real*)

Where we wish to discuss a number of functions all having the same domain and range it is convenient to abbreviate thus:

f, g, ... h all ∈ ... ⇒ ...

for

f ∈ ... ⇒ ...

and

g ∈ ... ⇒ ...

.....

and

h ∈ ... ⇒ ...

Some functions do not have a fixed number of arguments and some do not have a fixed number of results (*see* section 4.2 'Application of functions'). In such cases we may write for example

f ∈ *integer* ⇒ *real, integer, ... , integer*

for the domain or range, meaning that a real and a variable number of integers are the results.

2. ITEMS

2.1. Simple and compound items

The objects on which one can operate are called *Items*. They are divided into two distinct classes: *Compound* items, which are represented by addresses and *Simple* items which are directly represented by bitstrings which do not contain addresses (these bit strings are normally of fixed length for a given implementation, being a single machine word). The address representing a compound item points to a bit string whose length may vary from item to item. This bit string may contain other items. The areas of store immediately pointed to by two different compound items do not overlap.

The following standard function recognises compound items:

iscompnd ∈ *item* ⇒ *truthvalue*

Two kinds of simple item are distinguished: integers and reals. The following standard functions recognise them:

isinteger, isreal all ∈ *item* ⇒ *truthvalue*

The standard function = (an operation of precedence 7) is used to represent equality of items. For integers and reals it has the usual meaning. Its meaning for compound items is given in section 7.1 'Functions of data structures'.

= ∈ *item, item* ⇒ *truthvalue*

2.2. Integers

Integers are simple items. They may be positive, negative or zero. The size of the largest and smallest integers allowed depends on the implementation. The following functions on integers are standard:

intadd, intsub, intmult, all ∈ *integer, integer* ⇒ *integer*
// ∈ integer, integer ⇒ *integer, integer*
intplus, intminus all ∈ *integer* ⇒ *integer.*
intsign ∈ *integer* ⇒ *integer*
intgr, intle, intgreq, intleeq all ∈ *integer, integer* ⇒ *truthvalue*

intadd, intsub, intmult and *intdiv* are the usual add, subtract and multiply. *//* is divide with remainder and produces a quotient and a remainder (if $a//b$ is (q, r), then $q*b+r=a$ and $0 \leqslant r < b$). It is an operation of precedence 4.

intplus carries an integer into itself and *intminus* complements an integer. *intsign* produces -1, 0, or $+1$ according to the sign of the integer. The remaining four functions are the relations 'greater than', 'less than', 'greater than or equal to' and 'less than or equal to'.

The syntax of integers is:

⟨*integer*⟩::=⟨*octal integer*⟩|⟨*binary integer*⟩|⟨*decimal integer*⟩
⟨*octal integer*⟩::=8:⟨*octal digit*∗⟩
⟨*binary integer*⟩::=2:⟨*binary digit*∗⟩
⟨*decimal integer*⟩::=⟨*digit*∗⟩
⟨*octal digit*⟩::=0|1|2|3|4|5|6|7
⟨*binary digit*⟩::=0|1

Example:

⟨*integer*⟩*e.g.*::=8:777|2:101|0|6559

Integers may also be treated as bit-strings (the length depending on the implementation) and the following functions are standard:

logand, logor, logshift all ∈ *integer, integer* ⇒ *integer*
lognot ∈ *integer* ⇒ *integer*

logand and *logor* are the usual bit by bit 'and' and 'inclusive or'; *logshift* causes the first integer to be shifted left by the number of places given in the second, unless the second integer is negative when shifting to the right takes place (all new bits to fill up the end are zero in each case).

2.3. Reals

Reals are simple items. They may be positive, negative or zero. The size of the largest and smallest reals allowed and the precision depends on the implementation. The following functions on reals are standard:

realadd, realsub, realmult, realdiv all ∈ *real, real* ⇒ *real*
realplus, realminus all ∈ *real* ⇒ *real*
realsign ∈ *real* ⇒ *integer*
realgr, realle, realgreq, realleeq all ∈ *real, real* ⇒ *truthvalue*

These are the usual add, subtract, multiply and divide on reals. *realplus* carries a real into itself and *realminus* complements a real. *realsign* produces -1, 0 or $+1$ according to the sign of the real. The remaining four functions are the relations 'greater than', 'less than', 'greater than or equal to' and 'less than or equal to'.

There are also operations to convert a real to the nearest integer and to convert an integer to real:

intof ∈ *real* ⇒ *integer*
realof ∈ *integer* ⇒ *real*

The syntax of reals is as follows:

⟨*real*⟩::=⟨*decimal integer* ?⟩.⟨*decimal integer*⟩ ⟨*exponent* ?⟩
⟨*exponent*⟩::=$_{10}$+⟨*integer*⟩|$_{10}$−⟨*integer*⟩ | $_{10}$⟨*integer*⟩

Example:

⟨*real*⟩*e.g.*::=.5|1.99|1.5$_{10}$−6

2.4. Truth values

The two items *True* which is the integer 1 and *False* which is the integer 0 are called *Truthvalues.*

On entry to the POP-2 system the standard variable *true* is set to 1 and the standard variable *false* is set to 0. The following standard functions on truthvalues are provided:

booland, boolor all ∈ *truthvalue, truthvalue* ⇒ *truthvalue*
not ∈ *truthvalue* ⇒ *truthvalue*

These are the usual functions 'and', 'inclusive or' and 'not' of propositiona calculus.

2.5. Undefined

The standard variable *undef* has the word *"undef"* as its value on entry to the POP-2 system (*see* section 8.6 'Words'). The programmer may use it as the result of a function which fails to produce its normal result.

2.6. Terminator

The standard variable *termin* has the word *"termin"* as its value on entry to the POP-2 system (*see* section 8.6 'Words'). It may be used as the first argument of a variadic function (*see* section 4.2 'Application of functions') or to mark the end of an input file (*see* section 9.1 'Input').

3. VARIABLES

3.1. Identifiers

An item may be the *Value* of a *Variable* (a variable is not itself an item). An *Identifier* is associated with the variable and this identifier is used to

refer to it in a POP-2 program. A number of distinct variables may have the same identifier, but only one of them is *Currently associated* with it at a particular time in the evaluation process.

An identifier may be restricted to a certain range of values and it may be given special syntactic properties by being given a precedence (*see* section 5.2 'Precedence').

The syntax of identifiers is:

⟨*identifier*⟩::=⟨*letter*⟩ ⟨*alphanumeric* *?⟩ | ⟨*sign* *⟩
⟨*alphanumeric*⟩::=⟨*letter*⟩ | ⟨*digit*⟩

Example:

⟨*identifier*⟩*e.g.* ::=*x* | *y*99 | *alpha* | *u2a* | +++ | /+ | <> | * $ $ *

Syntax words such as **then, end,** → and : have special meanings and may not be used as identifiers. Only the first 8 characters are significant.

3.2. Declaration and initialisation

A variable is either *Global, Local* or *Formal.* A *Declaration* is used to introduce an identifier and associate it with a global or local variable. A *Local Declaration,* introducing a local variable, is a declaration which occurs in a function body. A *Global Declaration,* introducing a global variable, is one which does not.

An *Initialisation* is used to introduce an identifier and associate it with a formal variable and give the variable an initial value. It is achieved by including the identifier in the formal parameter list of a function (*see* section 4.1 'Definition of functions').

A declaration or initialisation may also specify that the identifier is restricted to take only functions as values. This is not necessary but may make the implementation more efficient. A declaration or initialisation may also specify that the identifier is an *Operation,* i.e. it is restricted to take functions as its values and is given a precedence. This restriction is associated with the unique name (see below) produced by the declaration or the initialisation.

The syntax of declarations is:

⟨*declaration*⟩::=**vars** ⟨*declaration list element* *⟩
⟨*declaration list element*⟩::=⟨*identifier*⟩ | ⟨*restriction*⟩
⟨*restriction*⟩::=⟨*restrictor*⟩⟨*identifier*⟩ | ⟨*restrictor*⟩(⟨*identifier* *⟩)
⟨*restrictor*⟩::=**function** | **operation** ⟨*integer*⟩

Example:

⟨*declaration*⟩*e.g.*::=**vars** *x y* | **vars** *x y* **function**(*f g*) **operation** 7 ==

A declaration or initialisation has a *Scope,* which is a piece of POP-2 text. An identifier may not be used to represent a variable outside the scope of a declaration or initialisation of the identifier.

The scope of a global declaration starts at the declaration and continues until the identifier is cancelled.

The scope of a local declaration starts at the declaration and continues to the end of the innermost function body enclosing it.

The scope of an initialisation is the body of the function in which it occurs.

Each declaration or initialisation gives rise to a unique mark and this mark is associated with all occurrences of any identifier introduced by the declaration or initialisation within the scope of the declaration or initialisation. An identifier together with its unique mark is called a *Unique name.*

Thus an identifier which occurs in more than one declaration or initialisation corresponds to more than one unique name.

The generation of fresh unique names for identifiers can be suppressed by using the standard routines:

nonunique, unique all ∈ () ⇒()

If *nonunique* is applied, all declarations or initialisations of a given identifier until *unique* is applied will give rise to the same unique name. This may save storage space and can be used when no confusion is liable to occur.

To sum up:

A new identifier is introduced by introducing a fresh sequence of characters.

A new unique name is introduced by each declaration or initialisation (unless *nonunique* has been applied).

A new variable is introduced by each dynamic activation of a declaration or initialisation.

A variable has an *Extent* which is a sequence of evaluations of expressions and statements.

The extent of a global variable starts from its declaration and continues indefinitely.

The extent of a local or formal variable starts on entry to the body of the function in which it is declared or initialised and continues until exit from the body. During this extent the extent of any other variable with the same unique name is temporarily interrupted. This is called a *Hole in the Extent* of the other variable. Its value is not altered but it cannot be accessed or changed by assignment. Thus there is only one variable *Currently Associated* with a particular unique name during any evaluation. Other variables associated with the unique name are in abeyance.

More than one global declaration of the same identifier is not permitted unless a cancellation of it intervenes in the text.

Similarly a declaration of a local variable is not permitted if there is already a declaration of a local or initialisation of a formal with the same identifier for the same function body.

A *Standard Variable* is a global variable which already has a value on entry to the POP-2 system. A *Standard Function* (or *Routine*) is one which is the value of a standard variable. Certain standard variables are *Protected*, i.e. no assignment may be made to them.

3.3. Cancellation

A cancellation terminates the scope of any declaration of an identifier and removes the effect of any restrictions placed upon the identifier. The cancellation must occur textually between the old declaration and any new declaration. It may not occur in a function body.

The syntax of cancellations is:

⟨*cancellation*⟩::=**cancel** ⟨*identifier* ∗⟩

4. FUNCTIONS

4.1. Definition of functions

A *Function* is a compound item. Definition and application of functions are treated in this section and the next. Certain properties of a function regarded as a data structure are treated in section 8.7 'Functions'.

A function consists of a *Formal Parameter List* which is a list of identifiers of formal variables, possibly an *output local list* which is a list of the identifiers of output local variables (*see* section 4.2 'Application of functions') and a *Body* which is an imperative sequence (*see* section 5.3 'Statements and imperatives').

A function which produces no results (*see* section 4.2 'Application of functions') is called a *Routine.*

Functions may be referred to in the program by using a function constant, called a *Lambda Expression*, or they may be standard functions provided by the POP-2 system, or they may be created by partial application or by application of a standard function which produces a function as a result.

The syntactic representation of a function constant is:

⟨*formal parameter list element*⟩::=⟨*ident*⟩ | ⟨*restriction*⟩
⟨*formal parameter list*⟩::=⟨*formal parameter list element* ∗?⟩
⟨*output local list element*⟩::=⟨*ident*⟩ | ⟨*restriction*⟩
⟨*output local list*⟩::= ⇒ ⟨*output local list element* ∗?⟩
⟨*function body*⟩::=⟨*imperative sequence*⟩
⟨*lambda expression*⟩::=**lambda**⟨*formal parameter list*⟩⟨*output local list*⟩;
⟨*function body*⟩ **end**

Example:

⟨*lambda expression*⟩*e.g.*::=**lambda** *x y*; *cons*(*x*, *cons*(*a*, *y*)) **end**
| **lambda** *x*; *nl*(1); *print*(*x*) **end**

We very often wish to declare a variable and then assign a function to it. The syntactic form of this will be as follows:

vars ⟨*identifier*⟩;
⟨*lambda*⟩⟨*formal parameter list*⟩; ⟨*function body*⟩ **end** → ⟨*identifier*⟩

This is so common that a special syntactic form is introduced which is equivalent to it:

⟨*function*⟩::=**function** | **routine**
⟨*function definition*⟩::=⟨*function*⟩⟨*identifier*⟩⟨*formal parameter list*⟩;
⟨*function body*⟩ **end**

The word **routine** is a synonym for **function**. It may be used for a function with no results.

If the identifier has been previously declared at this level no new declaration is implied and the function definition is equivalent simply to an assignment of a lambda expression. The identifier may be an operation identifier.

Example:

⟨*function definition*⟩*e.g.*::=**function** *max x y*; **if** $x > y$ **then** *x* **else** *y* **close**
end
| **routine** *enter u v*; *cons*(*conspair*(*u*, *v*), *dict*)→
dict **end**
| **function** *order x y* ⇒ *u v*;
if $x > y$ **then** $x \rightarrow u$; $y \rightarrow v$
else $y \rightarrow u$; $x \rightarrow v$ **close**
end

4.2. Application of functions

An *n-Tuple* is an ordered sequence of *n* items ($n \geqslant 0$). An item is identical with the 1-tuple whose sole member is that item. An *n*-tuple and an *m*-tuple may be *Concatenated* to produce an $(n+m)$-tuple.

A function of *n* arguments (i.e. with *n* formal parameters, excluding frozen formals; *see* section 4.4 'Partial application'), may be *Applied* to an *n*-tuple, whose members are called the *Actual Parameters* of the function. Application of a function to its actual parameters produces an *m*-tuple, whose members are said to be the *Results* of the function. A function producing no results (i.e. an 0-tuple) is called a routine (*see* section 4.1 'Definition of functions').

A function which does not take a fixed number of arguments is called *Variadic*. A function which does not produce a fixed number of results is called *Variresult*.

The application of a function to its actual parameters consists of the following sequence of events:

Entry: a new variable corresponding to each formal parameter is initialised to the corresponding actual parameter value, or if it is a frozen formal to the corresponding value in the frozen value list. A new variable corresponding to each local variable declaration in the function body but not in any interior function body is then created. The variables previously associated with the identifiers of formal or local variables can no longer be referred to but their values are undisturbed.

Running: the function body is evaluated with the variables created on entry.

Exit. Any items which have been placed on the stack (*see* section 5.3 'Statements and imperatives') and were not there at entry are concatenated with the values of any *Output Local Variables* to form the results of the function. The variables created on entry are terminated and the variable associated with each identifier reverts to what it was on entry. There is no change in the values of variables which were previously associated with the formal or local variable identifiers and have now been reinstated. The values of formal and frozen variables are lost. The frozen formals will be reinitialised from the frozen value list on the next entry to the function normally with the same values as last time; the frozen value list can be changed by using *frozval* (*see* section 8.7 'Functions').

4.3. Nonlocal variables

Variables which occur in a function body and are not locals (i.e. declared in the body) or formals (i.e. elements of the formal parameter list) are called *Nonlocal* to the function. They may be globals or locals of some outer function body which textually encloses it. Care must be taken not to apply a function with nonlocals in a hole in the extent of some of its nonlocals (*see* section 3.2 'Declaration and initialisation') or outside their extent. Mention of the identifier of such a nonlocal would refer to a quite different variable currently associated with that unique name. The difficulty can arise for recursive functions. Analogous trouble may arise if *nonunique* is used.

To avoid such difficulties a frozen formal may be used instead of the nonlocal, provided that it is not desired to assign a new value to the nonlocal as a result of the call. The frozen formal can be initialised by partial application to the value that the non-local would have taken. (Note that the frozen formals can be used in this way to give the equivalent of CPL fixed functions, *see CPL Reference Manual* privately circulated by C. Strachey, Programming Research Unit, Oxford University.) In cases where assignment to the nonlocal is desired a frozen formal can be used and initialised to take a reference (*see* section 8.1 'References') as value. The component of this reference can then be assigned to, and so long as the reference is made the value of some other exterior variable the value is accessible outside the function body.

4.4. Partial application

In section 4.2 'Application of functions' we explained the method of applying a function to its arguments. There is a process somewhat analogous to application called *Partial Application.* By this means some of the formal parameters of a function may be made into *Frozen Formals*, producing a new function with fewer arguments. The frozen formals are always initialised to a fixed value when the function is applied and do not require any corresponding actual parameters (*see* however section 8.7 'Functions' for means of altering this fixed value). In other words the actual parameters corresponding

to the frozen formals are supplied once and for all on partial application. The values of the frozen formals are called the *Frozen Value List.*

For example by partially applying the two argument function 'multiply' to 2 we get a one argument function to double a number, and by partially applying it to 3 we get a function to triple a number. These two functions can coexist, and in general one function can be used to generate any number of others by partial application.

More formally we say that a function f of m arguments may be partially applied to an n-tuple of actual parameters with $n \leqslant m$. We assume for the moment that f has no frozen formals. The partial application produces a new function f' with m-n ordinary formals corresponding to the first m-n formals of f, and n frozen formals corresponding to the last n formals of f. The function f' has a frozen value list consisting of the n items supplied as actual parameters of the partial application.

If f itself has some frozen formals already, say k of them, then f' will have $n+k$ frozen formals and $n+k$ corresponding items in its frozen value list.

The standard function *partapply* takes a function as its first argument and a list as its second argument, and partially applies the function to the elements of the list.

partapply ∈ *function, list* ⇒ *function*

Note that partial application constructs a new function with a particular frozen value list, it does not alter the original function in any way. A function which has been produced as the result of partial application is called a *Closure Function.* The frozen values of a closure function can be selected or updated (*see* section 8.7 'Functions').

If a doublet (*see* section 4.5 'Doublets') is partially applied to one or more items it produces a new doublet. The selector of the new doublet is obtained by partially applying the selector of the original doublet to the given items. The update routine of the new doublet is obtained by partially applying the update routine of the original doublet to the given items.

A special syntactic form is also available for partial application. It is similar to that for ordinary application (*see* section 5.1 'Expressions').

⟨*partial application bracket*⟩ = (%|%)
⟨*partial application*⟩ ::= ⟨*non-operation identifier*⟩ (% ⟨*expression list*⟩ %)
| ⟨*lambda expression*⟩ (% ⟨*expression list*⟩ %)

The value of the variable currently associated with the identifier is partially applied to the concatenation of the expressions in the expression list. Thus for example:

```
vars c; cons(%[is a number]%)→c;
c(1) ⇒
** [1 is a number]
c(2) ⇒
** [2 is a number]
```

```
function f x y z; .. etc. end;
f(% y1, z1 %)→f1; f1(x1)⇒
```

4.5. Doublets

When dealing with data structures, functions called selectors are defined which may be applied to a structure to produce its components (*see* section 7.1 'Functions of data structures'). To each selector there corresponds an update routine which alters the value of the component in the structure to a given new value.

Any function may have an update routine associated with it. This will normally only be done for selector functions. The function is then called a *Doublet*. When a function is created using a lambda expression its associated update routine is not defined. An update routine may be associated with it by using the doublet *updater*, (*see* section 8.7 'Functions').

When a variable whose value is a doublet is used as the operator of a compound expression the selector function of the doublet is applied. But when such a variable is used as the operator of a quasi compound expression (i.e. as part of a destination of an assignment) the update routine is applied.

It is convenient to extend our notation for functions (*see* section 1.5 'Notation for functions') using the new symbol '==⇒' to express concisely the domain and range of the selector and update routines of a doublet. Thus if f is a doublet we write

$$f \in d1, \ldots, dk ==\Rightarrow r$$

meaning that f has a selector s

$$s \in d1 \ldots, dk \Rightarrow r$$

and an update routine u

$$u \in r, d1, \ldots, dk \Rightarrow ()$$

Example:

The standard function *hd* used in list processing (*see* section 8.3 'Lists') is a doublet.

```
vars l; [1 2 3 4]→l; hd(l) ⇒
** 1
5→hd(l); l ⇒
** [5 2 3 4]
hd(l)⇒
** 5
function second l; hd(tl(l)) end;
lambda x l; x→hd(tl(l)) end→updater(second);
second(l) ⇒
** 2
6→ second(l); l ⇒
** [5 6 3 4]
```

4.6. Arithmetic operations

In sections 2.2 'Integers' and 2.3 'Reals' a number of standard functions were introduced for performing arithmetic on integers and reals.

We say that an item is a *Number* if it is either a real or an integer. Arithmetic on numbers is performed by the following standard operations:

Operation	*Precedence*	*Explanation*	*Result*
<	7	less than	truthvalue
>	7	greater than	truthvalue
=<	7	less than or equal	truthvalue
>=	7	greater than or equal	truthvalue
+	5	add	real or integer
−	5	subtract	real or integer
*	4	multiply	real or integer
/	4	divide	real
↑	3	exponent	real

These are defined in terms of *intadd*, *realadd*, etc. and *isreal*, *isint* and *realof*. +, − and * produce an integer result if both arguments are integer otherwise a real result.

5. EXPRESSIONS AND STATEMENTS

5.1. Expressions

An *Expression* is either a simple expression, a compound expression, a conditional expression or an imperative expression (*see* section 5.3 'Statements and imperatives').

A *Simple expression* is either an identifier or a *Constant*, a constant being an integer, a real or a structure constant. If the simple expression is an identifier then its value is the value of the variable currently associated with that identifier. If it is a constant then its value is the item denoted by the constant. A *Structure Constant* is either a lambda expression which is dealt with in section 4.1 'Definition of functions' and in section 8.7 'Functions', a word constant, a string constant or a list constant, all of which are dealt with in section 8 'Standard structures'.

A *Compound expression* has an *Operator* which is an expression and some *Operands* which are an expression list. The value of a compound expression is found by evaluating the operands and evaluating the operator, whose value should be a function (*see* section 4.5 'Doublets' for the case where the operator is a doublet). The sequence in which these evaluations are carried out is not defined. The function obtained from the operator is then applied to the *n*-tuple obtained by evaluating the operands. The case where the number of arguments required by the function is not equal to the number of items obtained by evaluating the operands is dealt with in section 5.3 'Statements

and imperatives'. The results of this application are the value of the expression. Thus the value of the expression is an n-tuple, with $n=0$ if the function is a routine.

Evaluation of conditional expressions is described in section 6.1 'Conditional expressions', and that of imperative expressions in section 5.3 'Statements and imperatives'.

An expression list is evaluated by evaluating the expressions of which it consists and concatenating the results. The order in which the evaluations are made is not defined. The order in which the results of evaluating the expressions are concatenated is the order in which the expressions occur.

The syntax of expressions is given below. There are a number of syntactic forms for compound expressions. A further explanation of the syntax is given in section 5.2 'Precedence'.

⟨*non-operation identifier*⟩::=⟨*identifier*⟩ | **nonop** ⟨*operation*⟩
⟨*constant*⟩::=⟨*integer*⟩ | ⟨*real*⟩ | ⟨*structure constant*⟩
⟨*structure constant*⟩::=⟨*lambda expression*⟩ | ⟨*quoted word*⟩ | ⟨*string constant*⟩ | ⟨*list constant*⟩
⟨*simple expression*⟩::=⟨*non-operation identifier*⟩ | ⟨*constant*⟩
⟨*operation*⟩::=⟨*identifier*⟩
⟨*parentheses*⟩::=(|)
⟨*compound expression*⟩::=⟨*non-operation identifier*⟩ (⟨*expression list*⟩)
|⟨*expression* ?⟩⟨*operation*⟩⟨*expression* ?⟩
|⟨*closed expression*?⟩⟨*dot operator**⟩
| ⟨*structure expression*⟩
⟨*closed expression*⟩::=⟨*simple expression*⟩ | ⟨*list expression*⟩
| ⟨*conditional expression*⟩
⟨*dot operator*⟩::= .⟨*non-operation identifier*⟩
⟨*structure expression*⟩::=⟨*partial application*⟩ | ⟨*list expression*⟩
⟨*expression list*⟩::=⟨*expression*?⟩ , ⟨*expression list*⟩ | ⟨*expression* ?⟩
⟨*expression*⟩::=⟨*simple expression*⟩ | ⟨*compound expression*⟩
| ⟨*conditional expression*⟩ | ⟨*imperative expression*⟩
| (⟨*expression list*⟩)

Examples:

⟨*simple expression*⟩*e.g.*::=*x* | **nonop**+ | 3 | **lambda** *x*; *x*+1 **end** | [3 5 9]
⟨*operation*⟩*e.g.*::= + | *** | *adjoin*
⟨*compound expression*⟩*e.g.*::=*f*(*x*+1, *y*) | *a**(*b*+*c*) | *x.hd*
| *f*(% *x* %) | [% *x*, *x*+1, *x*+2 %]
⟨*expression*⟩*e.g.*::=*a* | *g*(*h*(*x*+1)) | **if** *x*=0 **then** *y* **else** *z* **close**
| (*x*+1→*x*; *y*+1→*y*; *x***y*)
| (*x*, *y*+1, *z*−1)

The various syntactic forms of compound expressions denote the operator and operands in the following way:

(*i*) ⟨*non-operation identifier*⟩ (⟨*expression list*⟩). Here the operator is the identifier and the operands are the expression list.

(*ii*) ⟨*expression* ?⟩ ⟨*operation*⟩ ⟨*expression* ?⟩. This is equivalent to: **nonop** ⟨*operation*⟩ (⟨*expression* ?⟩, ⟨*expression* ?⟩) which is a special case of (*i*) above.

(*iii*) ⟨*closed expression* ?⟩. ⟨*non-operation identifier*⟩. This is equivalent to:
⟨*non-operation identifier*⟩ (⟨*closed expression* ?⟩) which is a special case of (*i*) above.

(*iv*) ⟨*structure expression*⟩. This is equivalent to (*i*) above with a special identifier for the operand. The exact rules are given in section 4.4 'Partial Application' and section 8.3 'Lists'.

Note that there is no syntactic provision above for compound expressions whose operator is an expression other than an identifier.

5.2. Precedence

If a compound expression or quasi compound expression is of the form

⟨*expression* ?⟩ ⟨*operation*⟩ ⟨*expression* ?⟩

the operator is the operation. In this case ambiguity might arise in the analysis of expressions such as

⟨*expression*⟩ ⟨*operation*⟩ ⟨*expression*⟩ ⟨*operation*⟩ ⟨*expression*⟩

which could be analysed with association to the left or to the right. This ambiguity is resolved by the notion of precedence. A precedence is a positive integer between 1 and 7 associated with an operation identifier. It is set by a declaration and can only be changed by cancellation. The operator of a sequence of expressions containing one or more operations is the operation of highest precedence or if there is more than one operation of highest precedence the rightmost of these.

It must be made clear that the difference between an operation and any other identifier which is restricted to having function values is purely a syntactic one.

It may be desired to use an operation in a context other than as the operator of a compound expression. If so it must be prefixed with the word **nonop** in which case it is treated syntactically like any other identifier. The use of **nonop** overrules the precedence of the identifier but does not remove restriction of its values to functions. This facility enables operations to appear as operands and enables assignment to operations.

Example:

> has precedence 7, + and − have precedence 5 and * has precedence 4.

$5-x+2*y>1+2$ is the same as $((5-x)+(2*y))>(1+2)$.

5.3. Statements and imperatives

A statement is either an assignment, a **goto** statement, a machine code instruction or an expression list. It may be labelled.

An imperative is either a declaration or a statement.
The syntax is:

⟨*statement*⟩::=⟨*assignment*⟩ | ⟨*goto statement*⟩
| ⟨*code instruction*⟩ | ⟨*expression list*⟩
| ⟨*labelled statement*⟩
⟨*imperative*⟩::=⟨*declaration*⟩ | ⟨*statement*⟩
⟨*imperative sequence*⟩::=⟨*imperative*⟩; ⟨*imperative sequence*⟩ |
⟨*imperative* ?⟩

Example:

⟨*imperative sequence*⟩*e.g.*::=*loop*:$x-1\rightarrow x$; $f(x)\rightarrow y$; **if** $x>0$ **then goto** *loop*;
| $x+1\rightarrow y$; y; $u\rightarrow y$;$\rightarrow z$;

The evaluation of an *Imperative Sequence* consists of evaluating the statements in the sequence in which they occur, except when a goto statement occurs and the sequence continues at the point indicated by the goto statement.

An *Imperative Expression* may be formed from an imperative sequence.
The syntax is:

⟨*imperative expression*⟩::=(⟨*imperative sequence*⟩)

The *Stack* is an ordered sequence of items. The last item to be added to this sequence is said to be on *Top of the Stack*. Items can be added to the top of the stack or removed from the top of the stack. On entry to the POP-2 system the stack is empty. When a statement is evaluated any results produced are added to the top of the stack. The results of an imperative sequence are the items left on the stack when the sequence has been evaluated.

Evaluation of a statement which is a compound expression may affect the stack as follows. If the number of arguments required by the function obtained by evaluating the operator is not the same as the number of items produced by evaluating the operands, these items are loaded on to the stack in sequence. The function then takes its arguments off the stack, the last argument being the one which was on the top of the stack. Thus suppose that the function requires m arguments and the operands yield n items. If $m>n$ the first $m-n$ arguments are taken off the stack. If $m<n$ the first $n-m$ items produced by the operands are left on the stack. If $m=n$ the stack is not affected by evaluating the compound expression. Exactly analogous remarks apply to quasi compound expressions.

5.4. Labels and goto statements

A *Label* may be attached to a statement. Evaluation of a *Goto Statement* using that label causes the sequence of evaluation to be changed so that the labelled statement is evaluated next. A goto statement may not refer to a label outside the function body in which it occurs. If a goto statement occurs

in an operand of a compound or quasi compound expression it may not refer to a label outside that operand. The syntax is:

⟨*labelled statement*⟩::=⟨*label*⟩:⟨*statement* ?⟩
⟨*goto statement*⟩::=**goto** ⟨*label*⟩ | **return**
⟨*label*⟩::=⟨*identifier*⟩

The statement **return** causes transfer of control to the exit of the innermost current function body. There is a standard macro **exit** which is synonymous with **return close.**

If an identifier or sign is used for a label it may not appear as an identifier associated with a variable in the text constituting that function body.

Goto statements and labelled statements may only occur inside a function body.

Note that a label is not an item.

Example:

loop: $x+1\rightarrow x$; $y*y\rightarrow y$;
if $x=0$ **then goto** *loop* **close.**

5.5. Assignment

An *Assignment* consists of a *Source*, which is an expression or sequence of expressions and a *Destination List*, which is a sequence of elements each of which is either an identifier or a *Quasi Compound Expression.*

A quasi compound expression has an operator which is an expression and some operands, i.e. a sequence of expressions (possibly an empty sequence). Note that a quasi compound expression is not an expression and cannot be evaluated alone to produce an item; it is merely a component of an assignment. It is syntactically the same as a compound expression but cannot be evaluated in isolation.

An assignment is evaluated as follows. First the source is evaluated to yield an n-tuple (where $n\geqslant k$, k being the number of destination elements). The last k elements of this n-tuple, which we will call the *Source Items*, are then taken in sequence starting from the last and each source item is combined with the corresponding destination element (taken in sequence starting from the first) as follows:

(*i*) If the destination element is a variable the source item becomes the new value of that variable.

(*ii*) If the destination element is a quasi compound expression the operator and operands of this expression are evaluated. The value of the operator must be a doublet (*see* section 4.5 'Doublets') and its update routine is applied to the concatenation of the source item and the values of the operands.

The syntax of quasi compound expression is given below. A further explanation of this syntax is given in section 5.2 'Precedence'.

⟨*quasi compound expression*⟩::=⟨*non-operation identifier*⟩ (⟨*expression list*⟩)

| ⟨*expression* ?⟩ ⟨*operation*⟩ ⟨*expression* ?⟩
| ⟨*closed expression* ?⟩
⟨*dot operator* * ?⟩

The syntax of assignments is:

⟨*assignment*⟩::=⟨*expression list*⟩ ⟨*destination* *⟩ | ⟨*function definition*⟩
| ⟨*macro definition*⟩
⟨*destination*⟩::= → ⟨*non-operation identifier*⟩ | → ⟨*quasi compound expression*⟩

Example:

⟨*assignment*⟩*e.g.*::=$x+1 \rightarrow y \mid u+v \rightarrow a(i,j) \mid x//y \rightarrow u \rightarrow v$

In the second example $a(i,j)$ is a quasi compound expression and the whole assignment is a euphemism for $a1(u+v, i, j)$ where $a1$ is the update routine of the doublet a.

Function definitions and macro definitions are special syntactic forms for assignments.

5.6. Comments

The word **comment** and all characters after it up to and including the next semicolon are ignored.

6. CONDITIONALS

6.1. Conditional expressions

A conditional expression is composed of three components which we will call the *Condition*, the *Consequent* and the *Alternative*. The condition is an expression with a single result, a conjunction or a disjunction (*see* section 6.2 'Conjunctions and disjunctions'). The consequent and the alternative are imperative sequences each having the same number of results. The method of evaluation of a conditional expression is as follows:

The condition is first evaluated. If its value is the truth value *true* then the consequent is evaluated and its value becomes the value of the expression. But if the value of the condition is the truth value *false* then the alternative is evaluated and its value becomes the value of the expression.

The alternative of a conditional expression may be omitted if it is an empty imperative sequence.

It will often happen that the alternative is itself a conditional expression. The syntax of conditionals is arranged to provide a compact notation to express this:

⟨*conditional body*⟩::=⟨*imperative sequence*⟩
⟨*conditional expression*⟩::=**if** ⟨*condition*⟩ **then** ⟨*conditional body*⟩
⟨*elseif clause* * ?⟩ ⟨*else clause* ?⟩ **close**

⟨*elseif clause*⟩::=**elseif** ⟨*condition*⟩ **then** ⟨*conditional body*⟩
⟨*else clause*⟩::=**else** ⟨*conditional body*⟩

Example:

⟨*conditional expression*⟩*e.g.*::=**if** $x>0$ **and** $x<3$ **then** y **elseif** $x>3$ **then** z
else 0 **close**
| **if** $x=0$ **then** $1\rightarrow y$ **close**

If there are no elseif clauses the conditional body is the consequent and the else clause is the alternative (which may be omitted). If there are elseif clauses then the first expression is the condition, the second is the consequent and the remainder is the alternative, and it is to be regarded as the conditional expression obtained by replacing the first **elseif** by **if** and inserting an extra **close** before the **close**, e.g.

if p **then** x **elseif** q **then** y **else** z **close**

is equivalent to

if p **then** x **elseif** q **then** y **else** z **close close**.

6.2. Conjunctions and disjunctions

A *Conjunction* is composed of two component expressions each producing a single result. The method of evaluating a conjunction is to evaluate the first component expression and if its value has the truth value *false* the value of the conjunction has truth value *false*, otherwise the second expression is evaluated and the conjunction has a truth volue equal to that of the second component expression.

A *Disjunction* is composed of two component expressions each producing a single result. The method of evaluating a disjunction is to evaluate the first component expression and if its value has the truth value *true* the value of the disjunction has truth value *true*, otherwise the second expression is evaluated and the disjunction has a truth value equal to that of the second component expression.

A number of conjunctions and disjunctions can be combined to form a condition.

The syntax is:

⟨*condition*⟩::=⟨*expression*⟩ **and** ⟨*condition*⟩ | ⟨*expression*⟩ **or** ⟨*condition*⟩
| ⟨*expression*⟩

These three kinds of conditions are respectively a conjunction, a disjunction and an expression.

Thus **and** and **or** associate to the right.

⟨*condition*⟩*e.g.* ::=$x<10$ **and** $x>0$ | $x>10$ **or** $x<0$ | *null*(x) | b
| $p(x)$ **and** $q(x)$ **or** $r(x)$

In the last example the following cases can occur ('–' means that the expression is not evaluated)

$p(x)$	$q(x)$	$r(x)$	value of condition
false	—	—	*false*
true	*false*	*false*	*false*
true	*false*	*true*	*true*
true	*true*	—	*true*

7. DATA STRUCTURES

7.1. Functions of data structures

A *Data Structure* is a compound item which has other items as its *Components*. For each class of data structures there is a family of functions called the *Characteristic Functions* acting upon structures of that class. These functions are a constructor, a destructor, selectors and update routines. A given compound item may represent a number of different data structures by being used in association with more than one family of functions and hence having different components.

Given values for its components it is possible to construct a data structure using a *Constructor* function, say *c*.

c ∈ *component*, ... , *component* ⇒ *data structure*

It is possible to select the value of a component of a data structure. For each component there is a *Selector* function, say *si*.

si ∈ *data structure* ⇒ *component*

It is possible to update a component of a data structure, i.e. to give it a new value. For each component there is an *Update Routine*, say *ui*:

ui ∈ *component*, *data structure* ⇒ ()

When a data structure is updated the old version is overwritten.

It is convenient to define another function called a *Destructor* function, say *d*, which is the inverse of the constructor, i.e. given a data structure it produces its components as results.

d ∈ *data structure* ⇒ *component*, ... , *component*

After applying the destructor to a structure, the structure is deleted.

There is a relation called equality (*see* section 2.1 'Simple and compound items') which may hold between two compound items. It is denoted by the standard function = (an operation of precedence 7). This function is also defined for simple items with the usual meaning.

= ∈ *item*, *item* ⇒ *truthvalue*

Thus if the value of an expression '*E*1' is equal to the value of an expression '*E*2' then the expression

*E*1 = *E*2

has value *true*.

Equality means that the two compound items contain the same address, i.e. they point to the same area of store. If the items are not equal they point to entirely different areas of store. We say that a compound item is *Copied at the Top Level* if a new item is formed pointing to a new area of store which contains items equal to those of the given compound item. The new item and the previous one are not equal. They are however *Equivalent.*

Equivalent compound items are defined as items which are either equal or all of whose components are equivalent.

Updating an item alters a component item in the store area pointed to by the item but does not cause copying.

We will now give a more formal explanation of equality, but the model in terms of addresses and storage may be kept in mind.

Equality is an equivalence relation, i.e. it is

(*i*) reflexive ($x=x$);
(*ii*) symmetric (if $x=y$ then $y=x$); and
(*iii*) transitive (if $x=y$ and $y=z$ then $x=z$).

It has the following other properties:

(*iv*) The value of a formal parameter variable is equal to the corresponding actual parameter.

(*v*) If an item is assigned to a variable then the value of the variable is equal to that item.

(*vi*) An item, other than a word or simple item, which is read in (*see* section 9.1 'Input') is not equal to any other item.

(*vii*) The rules for equality of words are given in section 8.6 'Words'.

(*viii*) Two integers or two reals are equal according to the usual rules of arithmetic. An integer is never equal to a real.

Items are equal only if their equality follows from the above properties.

We can now state some relationships between the various functions on data structures. We will use a and b for data structures, $x1, \ldots, xi, \ldots, xk$ for items occurring as components, $s1, \ldots, si, \ldots, sk$ for selectors, $u1, \ldots, ui, \ldots, uk$ for update routines, c for a constructor and d for a destructor.

(*i*) $s1(a), \ldots, sk(a)$ is the same n-tuple as $d(a)$, i.e. they have equal elements.

(*ii*) $si(c(x1, \ldots, xi, \ldots, xk))=xi$ is true.

(*iii*) $c(s1(x), \ldots, sk(x))=x$ is always false, but the left-hand expression is equivalent to x.

(*iv*) After evaluating $ui(xi, a)$,

$si(a)=xi$ is true.

(*v*) After evaluating $ui(si(a), a)$, a is unchanged.

(*vi*) If $a=b$ then $ui(xi, a)$ is evaluated,

$si(b)=xi$ is true and $a=b$ is still true.

(*vii*) From (*i*) and (*ii*) above $d(c(x1, \ldots, xk))$ is the same n-tuple as $x1, \ldots, xk$, i.e. they have equal elements.

If a and b are data structures and a is not equal to b and updating a com-

ponent of *a* also updates some component of *b* then *a* and *b* are said to *Share*.

When we wish to discuss a class of data structures which do not all have the same number of components (such as strips, *see* section 7.3 'Strips') it is convenient to define a *General Selector* function and a *General Update Routine*.

The general selector function, say *s*, has as arguments, an integer, *i*, and a data structure. It selects the *i*th component of the data structure.

s ∈ *integer, data structure* ⇒ *component*

Thus if *si* is the *i*th selector $s(i, a) = si(a)$.
Similarly for the general update routine, say *u*,

u ∈ *component, integer, data structure* ⇒ ( )

Thus if *ui* is the *i*th update routine, $u(xi, i, a)$ has the same effect as $ui(xi, a)$.

The programmer is able to create new kinds of data structures called records and strips (*see* section 7.2 'Records' and 7.3 'Strips'). He can also create functions by methods already described. He may be able to create other kinds of data structures using extra standard functions or machine code but this depends on the implementation. Certain classes of records and strips are standard and these are described in section 8 'Standard structures'.

There are a number of special expressions called 'structure expressions' used to construct these standard structures (*see* section 5.1 'Expressions').

Given a class or several classes of data structures with their associated functions it is possible to define functions which characterise a new family of data structures. Suppose for example that we have a class of structures with two selectors, say *s*1 and *s*2, and components which are full items and members of the same class of structures. We can then define a new class of structures whose selectors are given by:

function *s*11 *a*; *s*1(*s*1(*a*)) **end**; **function** *s*12 *a*; *s*1(*s*2(*a*)) **end**;
function *s*21 *a*; *s*2(*s*1(*a*)) **end**; **function** *s*22 *a*; *s*2(*s*2(*a*)) **end**;

If *c* is the constructor of the first class we define the new constructor:

function *cc x*1 *x*2 *x*3 *x*4; *c*(*c*(*x*1, *x*2), *c*(*x*3, *x*4)) **end**;

Note that if it is associated with two or more families of functions the same compound item can represent two or more structures, one of each class. However, for each class of compound items there is one *Primitive Data Structure Class* and other data structures are defined in terms of this primitive class. A primitive data structure does not share with any other primitive data structure.

7.2. Records

A *Record* is a compound item which is a member of a *Record Class*. The *Size* of a set of items is an integer item. If all the items in the set are restricted

to be non-negative integers less than 2^n, the size is the integer n, otherwise if the component is a *Full Item* (i.e. the set is not restricted) the size is the integer 0. For each component of a record there is a size associated with the set of possible values of that component. The *Specification of a Record* is the list of sizes associated with its components. A record class is a set of records which all have the same specification, and this is said to be the specification of the record class. Note that a record class is not an item. A word is associated with each record class.

A family of functions is associated with each record class to form a primitive class of data structures. This family comprises a set of selectors (∈ *record* ⇒ *component*) and a set of corresponding update routines (∈ *component*, *record* ⇒ ( )), a constructor (∈ *component*, ... , *component* ⇒ *record*) and a destructor (∈ *record* ⇒ *component*, ... , *component*). Each selector function may be paired with the corresponding update routine to form a doublet (∈ *record* =⇒ *component*). The standard function *recordfns* is used to create a new record class. It requires as arguments the word to be associated with the record class, an estimate of the number of records in the record class (this is purely to help in efficient implementation) and the specification of the record class. It produces the constructor, the destructor and the doublets for the record class. The number of its results depends on the length of the specification list. Normally the programmer will immediately assign these resulting functions to variables.

recordfns ∈ *word*, *integer*, *specification* ⇒ *constructor*, *destructor*, *doublet*, ... , *doublet*

There is a standard function which converts a record to a list of its components:

datalist ∈ *record* ⇒ *list*

There is a function *dataword* which given a record produces the word associated with its record class.

dataword ∈ *record* ⇒ *word*

The function *copy* copies a record at the top level.

copy ∈ *record* ⇒ *record*

The functions *datalist*, *dataword* and *copy* are defined over records of any class, and whenever *recordfns* is used to create a new record class these three functions are extended to deal with records of that class.

The routine *enddata* may be given the word associated with a record class and removes all records in that class. It also adjusts the three functions just mentioned so that they no longer deal with that record class.

enddata ∈ *word* ⇒ ()

7.3. Strips

A *Strip* is a compound item which is a member of a *Strip Class*. All components of a strip must have the same size (*see* section 7.2 'Records'

for definition of size) which is called the *Component Size* of the strip. Al strips in a strip class must have the same component size but not necessarily the same number of components. A word is associated with each strip class.

A family of functions is associated with each strip class to form a primitive class of data structures. This family includes a general selector function (∈ *integer, strip* ⇒ *component*) and a general update routine (∈ *component, integer, strip* ⇒ ()). The selector function may be paired with the update routine to form a doublet. It also includes for each strip class an initiator function (∈ *integer* ⇒ *strip*). This constructs a strip with the given number of components, but the values of these components are not defined. The initiator may be used with the update function to define a constructor function for strips of the strip class.

The standard function *stripfns* is used to create a new strip class. It takes as arguments the word to be associated with the strip class, an estimate of the total number of all components of all strips in the strip class (this is purely to help in efficient implementation) and the component size of the strip class. It produces as results the initiator function and the doublet for the strip class:

stripfns ∈ *word, integer, size* ⇒ *initiator, doublet*

There is a standard function which converts a strip to a list of its components. This is *datalist* (*see* section 7.2 'Records'). There is a function which given a strip produces the word associated with its strip class. This is *dataword* (*see* section 7.2 'Records').

The function *copy* copies a strip at the top level (*see* section 7.2 'Records').

The functions *datalist, dataword* and *copy* and the routine *enddata* act for strips just as for records.

7.4. Garbage collection

Storage for the construction of data structures is made available by a storage control system. This system must be able to make use of areas of store which have been used but are no longer required. This is achieved by a process known as *Garbage Collection* which is undertaken whenever the system runs short of store. This first of all discovers what items can still be referred to by the programmer, e.g. because they are the value of a variable whose extent has not finished (*see* section 3.2 'Declaration and initialisation'). Any items which can no longer be referred to are destroyed, i.e. their storage area is returned to the system for use in constructing other items. Since he cannot refer to them the programmer is not aware of this destruction.

If variables refer to compound items which are no longer in use, the garbage collector cannot recover the associated storage. The variable should be reset, e.g. to zero. In the case of identifiers the identifier can be cancelled (*see* section 3.3 'Cancellation').

To avoid too frequent garbage collection compound items can be deleted, i.e. returned to the storage control system, using the standard routine:

delitem ∈ *item* ⇒ ( )

When an item is deleted its components are not deleted.

After an item has been deleted it is no longer available and the onus is on the programmer not to use it. The implementation may not give an error message if he does use it, the value simply not being defined.

8. STANDARD STRUCTURES

8.1. References

There is a standard record class called *References*. These have one component which is a full item. The word associated with the class is *"ref"*. Thus before entry to the POP-2 system this class is created using *recordfns*, and the resulting functions are assigned to variables to give the following standard functions:

constructor: *consref* ∈ *item* ⇒ *reference*
destructor: *destref* ∈ *reference* ⇒ *item*
doublet: *cont* ∈ *reference* ==⇒ *item*

A reference may be used e.g. as an actual parameter of a function to enable the function to cause side effects by updating the reference.

8.2. Pairs

There is a standard record class called *Pairs*. Records of this class have two components which are both full items. The word associated with the class is *"pair"*. Thus before entry to the POP-2 system this class is created using *recordfns*, and the resulting functions are assigned to variables to give the following standard functions:

constructor: *conspair* ∈ *item*, *item* ⇒ *pair*
destructor: *destpair* ∈ *pair* ⇒ *item*, *item*
doublets: *front*, *back all* ∈ *pair* ==⇒ *item*

An *Atom* is an item which is not a pair. Atoms are recognised by the standard function *atom*.

atom ∈ *item* ⇒ *truthvalue*

8.3. Lists

There is a standard data structure called a *Link* which is used to construct another data structure called a *List*. Lists in POP-2 include structures analogous to LISP lists, but also structures which compute the elements dynamically (cf. P. J. Landin's 'streams').

The word *"nil"* is used to represent the *Null List* and the standard variable

nil takes this value on entry to the POP-2 system. The null list is also represented by a pair whose front is *true* and whose back is a function.

The standard function *null* recognises the null list

null ∈ *list* ⇒ *truthvalue*

A list is either the null list or it is a link.

A link is either:

(*i*) a pair whose front component is any item and whose back component is a list, or

(*ii*) a pair whose front component is *false* and whose back component is a function with no arguments and one result.

In case (*ii*) the function is one which when repeatedly applied produces a succession of items, not necessarily all the same, i.e. normally the function will side-effect itself. The last item produced should be the terminator. For example this enables us to convert an input file to a list. Lists with this sort of link are dynamic and some or all of their elements are computed rather than stored statically.

The characteristic functions of a link are:

constructor: *cons* ∈ *item*, *list* ⇒ *link*
destructor: *dest* ∈ *list* ⇒ *item*, *list*
doublets: *hd* ∈ *link* ==⇒ *item* (called the 'head')
tl ∈ *link* ==⇒ *item* (called the 'tail')

These functions are very similar to those for pairs, but in the case of a link of the second kind special precautions are taken to make sure that on applying the selector *tl* the front component is not lost but preserved in a pair. Thus if x has a list as its value and *tl*(x) is evaluated there is a side effect on x, but matters are so arranged that this side effect is not detectable using the list processing functions. The function *cons* is the same as *conspair* and produces a link of the first kind. The following standard function produces a link of the second kind or the null list given a function of no arguments:

fntolist ∈ (() ⇒ *item*) ⇒ *list*
function *fntolist f*; *cons*(*false*, *f*) **end**

The other characteristic functions are defined as follows:

First an auxiliary function (*not* standard) to convert the first link of a dynamic list to static form.

```
function solidified 1; vars f x;
  if isfunc(back(1))
    then back(1)→f; f() → x;
      if x=termin then true→front(1)
        else x→front (l); conspair(false, f)→back(l)
      close; l
    else 1
  close
end;
```

```
function hd l; front(solidified(l)) end;
lambda i l; i→front(solidified(l)) end→updater(hd);

function tl l; back(solidified(l)) end;
lambda i l; i→back(solidified(l)) end→updater(tl);
function dest l; vars f;
  if isfunc(back(l)) then back(l)→f; (f(),l)
    else (front(l),back(l))
  close
end;
function null l;
  if l=nil then true
    elseif isfunc(back(l))
      then if hd(l) or null(solidified(l)) then true else false close
    else false
  close
end;
```

A list may have no components (if it is the null list) or one or more (if it is a link).

If it is a link its first component is the head component of the link and its remaining components are the components of the list which is the tail component of the link. Thus the characteristic functions for lists can be defined in terms of those for links.

There are two special syntactic forms for constructing lists. These are list constants and list expressions. List constants may have lists, integers, reals, words or strings (see section 8.4 'Full strips and character strips') as components. The list is constructed at compile time.

⟨*list constant brackets*⟩::=[|]
⟨*list constant*⟩::=[⟨*list constant element* *?⟩]
⟨*list constant element*⟩::=⟨*list constant*⟩ | ⟨*character group*⟩

Example:

⟨*list constant*⟩*e.g.*::=[1 2 *DOG CAT*] | [[1 2] [4 5] 6]

List expressions are formed by evaluating a number of expressions at run time and constructing a list.

⟨*list expression brackets*⟩::=[% | %]
⟨*list expression*⟩::=[%⟨*expression list*⟩%]

Thus
[% %] is equivalent to *nil*
and [% ⟨*expression*⟩ %] is equivalent to *cons*(⟨*expression*⟩, *nil*)
and [% ⟨*expression*⟩, ⟨*expression list*⟩ %] is equivalent to
cons(⟨*expression*⟩, [%⟨*expression list*⟩%])

Example:
⟨*list expression*⟩*e.g.*::=[%*x*+1, [%*x*+2, *x*+3%], *tl*(*y*)%]

For convenience the following functions are standard:

next ∈ *list*⇒*item*, *list* (similar to *dest* but non-destructive)
::(a synonym for *cons* but an operation of precedence 2)
⟨⟩ ∈ *list*, *list*⇒ *list* (concatenates the lists, an operation of precedence 2).

8.4. Full strips and character strips

Two strip classes are standard.

The first is *Full Strips* with full items as components and associated word *"strip"*. The characteristic functions are:

initiator: *init* ∈ *integer* → *full strip*
doublet: *subscr* ∈ *integer*, *strip* ==⇒ *item*

The second is *Character Strips* (also called '*Strings*') (for characters *see* section 8.6 'Words') with component size 6 and associated word *"cstrip"*. The characteristic functions are:

initiator: *initc* ∈ *integer* ⇒ *character strip*
doublet: *subscrc* ∈ *integer*, *strip* ==⇒ *integer of size* 6

The components of a character strip may be any integers of size not more than 6, they need not necessarily be used to represent characters.

There is a structure constant to construct character strip constants at compile time.

⟨*string bracket*⟩::=' | `
⟨*string constant*⟩::='⟨*string constant element* * ?⟩`
⟨*string constant element*⟩::=⟨*string constant*⟩ | ⟨*any character except a string bracket*⟩

Example:

⟨*string constant*⟩*e.g.*::='... *rubbish . please type* 'sorry``

Spaces and newlines are significant in string constants.

There are functions to input and output character strings stored as character strips (*see* section 9.1 'Input and output'). The external format is as for string constants.

8.5. Arrays

Arrays give a convenient method of accessing and updating structures indexed by integers. An array has components, which are items of a given size. Each component is associated with a sequence of integers called *Subscripts*. The number of subscripts is known as the number of *Dimensions* of the array. An array is a doublet:

array ∈ *subscript*, ..., *subscript* ==⇒ *component*

This is in contrast to strips which have a general selector and a general update routine associated with a whole class of strips and take the actual strip referred to as a parameter. Arrays can be formed from strips (or from other data structures) by using partial application. The programmer is free

to do this in any way he chooses but standard functions for creating arrays are provided.

There is a standard function to create a many dimensional array of items of any size. Updating a component of this array does not affect any other component. This function is:

newanyarray ∈ *boundslist*, (*subscript*, ... , *subscript* ⇒ *component*),
strip initiator, *strip doublet* ⇒ *array*

The array produced will normally be immediately assigned to a variable.

The boundslist is a list of integers, these two integers being alternately the lower and upper bounds for each subscript. The second parameter is a function used to initialise the components of the array. It must produce the appropriate component for each combination of subscripts. The strip doublet and strip initiator are the characteristic functions of a strip class whose components are of the same size as that required for the array components.

There is also a standard function to create arrays of full items:

newarray ∈ *boundslist*, (*subscript*, ... , *subscript* ⇒ *component*) ⇒ *array*

This is obtained by partial application and is equivalent to

newanyarray (% *init*, *subscr* %).

8.6. Words

There is a standard record class called *Words*. It has 8 components of size 6 called *Characters*, and a component called the *Meaning*. The word associated with the record class is *"word"*. The standard functions characterising words are:

constructor: *consword* ∈ *character*, ... , *character*, *integer* ⇒ *word*
destructor: *destword* ∈ *word* ⇒ *character*, ... , *character*, *integer*, *item*
doublets: *charword* ∈ *word* ==⇒ *character*, ... , *character*, *integer*
meaning ∈ *word* ==⇒ *item*

Each character of the POP-2 character set corresponds to a unique integer. The correspondence rule depends on the implementation. Note that the functions above are variadic and work on a variable number of characters followed by that number as an integer. If there are less than 8 characters supplied to the constructor the remaining character components are not defined and they are not produced by the destructor or selector. The constructor does not take a meaning component as argument. The meaning of a word is *undefined* unless the word has been updated to have a particular meaning.

Words may occur in the program as quoted words, i.e. word constants, with the following syntax:

⟨*unquoted word*⟩::=⟨*letter*⟩ ⟨*alphanumeric* *?⟩ | ⟨*sign* *⟩
| ⟨*decorated bracket*⟩ | ⟨*bracket decorator*⟩
| ⟨*separator*⟩ | ⟨*period*⟩ | ⟨*exponent*⟩ | ⟨*quote*⟩

⟨*decorated bracket*⟩::=(|) | (% | %)
|[|] | [% | %]

⟨*quoted word*⟩::="⟨*unquoted word*⟩"

Example:

⟨*quoted word*⟩*e.g.*::="*big*" | "++" | "%)" | """

Words may also occur as components of constant lists (*see* section 8.3 'Lists'). Only the first 8 characters are significant.

Words may also be read as data (*see* section 9.1 'Input'). Words which occur as constants or are read as data are *Standardised*, i.e. if a word with the same characters already exists no new word is constructed and the compound item produced is the previously existing word, but if no such word exists a new word is constructed with *undef* as its meaning. Words constructed using *consword* are also standardised, but the update routines do not standardise.

8.7. Functions

Functions are compound items. There is no constructor or destructor for functions. They can be constructed by the methods described in section 4.1 'Definition of functions', and they can be deleted by the routine *delitem* (*see* section 7.4 'Garbage collection'). There is a family of characteristic functions associated with the class of functions to form a primitive class of data structures.

Functions have an accessible component which may be used to associate extra information with the function. It is accessed by the standard doublet

fnprops ∈ *function* ==⇒ *item*

Functions have an update routine (*see* section 4.5 'Doublets'). For a function constructed by using **lambda** or **function** or **routine** this has initially no defined value. This component may be selected or updated by using a standard doublet:

updater ∈ *function*==⇒*routine*

Closure functions i.e. those constructed by partial application, have a doublet to select or update the values of their frozen formals

frozval ∈ *integer*, *closure function*==⇒*item*

The integer determines which of the frozen formal values is affected, counting from the front (if a closure function is obtained by successive partial applications only the formals frozen by the last one are counted). There is also a doublet to select the function from which the closure function was constructed or replace it with another function.

fnpart ∈ *closure function*==⇒*function*

The standard function = follows the usual rules for compound items when applied to functions, i.e. equality is preserved over assignment, updating and actual parameter/formal parameter correspondence but each construction of a function produces a different one.

The following standard function recognises functions:

isfunc ∈ *item* ⇒ *truthvalue*

9. INPUT AND OUTPUT

9.1. Input

Information which is input to the POP-2 system is organised into *Files*, each of which comes from a *Device*.

Before a file can be accessed it must be *Opened*. From then on it can be read one character at a time. Eventually it must be *Closed*.

The naming of files and devices depends on the operating system of the implementation. The names of files are lists and the names of devices may be any item. A device name may refer to more than one device.

There is a standard variresult function *popmess* used for communicating with the operating system

popmess ∈ *list*⇒*item*, . . , *item*

This is used for various input and output purposes.

To open a file from a given device, *popmess* is used to produce a function to read characters from it i.e. a function ∈ ()⇒*character*. The list supplied to *popmess* has a head which is an input device name and a tail which is a file name.

To close a file before reaching the end of it, *popmess* is again used. The list supplied to it has a head which is the word *"close"* and a tail which is a list of one element: a character reading function obtained when the file was opened. No result is produced by *popmess* in this case.

The sequence of characters making up a POP-2 text may be split up into *Character Groups* each of which represents a *Text Item*. A text item is either an integer, a real, a word or a string. It is represented by a character group, thus:

⟨*character group*⟩::=⟨*integer*⟩ | ⟨*real*⟩ | ⟨*unquoted word*⟩
| ⟨*string constant*⟩

Character groups are terminated by spaces or newlines where necessary to separate them from the following character group.

There is a standard function to convert a function which produces a character whenever it is applied into a corresponding one which produces a text item whenever it is applied.

incharitem ∈ (() ⇒ *character*) ⇒ ((*l*) *text item*)

The program is input on a standard file called the *Standard Input File* from a standard device called the *Standard Input Device*. There is a standard function to read characterx from the *standard input* file:

charin ∈ () ⇒ *character*

The program is compiled from the text item list which is the value of the standard variable *proglist*. Initially this has as value the list of text items from the standard input file. It may be assigned to by the programmer who wishes to compile from a different source.

For convenience there is a standard function *itemread* producing the next item of the list which is the value of *proglist*.

itemread ∈ () ⇒ *text item*

It is defined thus:—

function *itemread*; *proglist* . *dest* → *proglist* **end**;

9.2. Output

Information which is output from the POP-2 system is organised into files, each of which is sent to a device (*see* section 9.1 'Input').

To open an output device the standard function *popmess* (*see* Section 9.1 'Input') is used to produce a routine to deliver characters to it i.e. a routine ∈ *character* ⇒ (). The list supplied to *popmess* has a head which is an output device name and a tail which is a file name.

There is a standard function to convert a routine which delivers a sequence of characters to an output file into one which delivers a sequence of text items.

outcharitem ∈ (*character* ⇒ ( )) ⇒ (*text item* ⇒ ( ))

Compiler messages and results of computation are normally output on a standard file called the *Standard Output File* to a standard device called the *Standard Output Device*. There is a standard routine to output characters to the standard output file:

charout ∈ *character* ⇒ ( )

There is a standard variable *cucharout* which contains the routine to output characters to the *Current Output File*. This contains initially the routine for the standard output file but it may be assigned to if a different output file is to be made current. An output file is closed by outputing the terminator.

There are standard routines to output spaces or newlines to the current output file:

sp ∈ *integer* ⇒ ( )
nl ∈ *integer* ⇒ ( )

There is a standard function which outputs any item to this file in some suitable format and produces that item unchanged as its result.

print ∈ *item* ⇒ *item*

There is a standard macro which uses *print* and causes the items on the stack starting at the bottom to be printed on a newline preceded by two asterisks. These items are removed from the stack. In a function body only the top item of the stack is affected. This macro is denoted by the POP-2

identifier ⇒ (not to be confused with the ⇒ used in this manual to show the type of functions). A semicolon is implied before and after so that immediate evaluation can occur (*see* section 11.1 'Immediate evaluation').

10. MACHINE CODE

It is possible to insert sections of machine code in an imperative sequence. The rules depend on the implementation. A code instruction is represented by the identifier $ followed by any sequence of characters which do not include ';'

⟨*code instruction*⟩::=$⟨*any sequence of characters other than* ;⟩

11. MODES OF EVALUATION

11.1. Immediate evaluation

A POP-2 program consists of a sequence of imperatives and cancellations:

⟨*program element*⟩::=⟨*imperative*⟩ | ⟨*cancellation*⟩
⟨*program*⟩::=⟨*program elementt*⟩;⟨*program*⟩

The program elements are evaluated in sequence in the same way as an imperative sequence. Each program element is evaluated as soon as the terminal semicolon and a space has been read by the compiler. The body of any function in the program element will be compiled and kept so that it may be evaluated when that function is applied.

11.2. Macros

A *Macro* is a routine which is applied at compile time.

The definition of a macro routine is similar to that of any other routine except that **macro** is used instead of **routine** and no formal parameters are allowed.

⟨*macro definition*⟩::=**macro** ⟨*identifier*⟩; ⟨*function body*⟩ **end**

A macro, like an operation, is applied whenever it is mentioned and does not need parentheses after it.

Although a macro has no parameters the function *itemread* (*see* section 9.1 'Input') may be used to read the text items following the macro identifier. There is a standard routine which, when applied in a macro body to a list of text items, concatenates these items to the right of the macro identifier in the program sequence of text items.

macresults ∈ *text item list* ⇒ ()

If it is applied more than once it concatenates to the right of the previously inserted items. On exit from the macro the inserted text items are evaluated as program.

Example:

```
macro --→; vars x y; itemread → x; itemread → y;
        macresults ([% "→", y, "→", x %])
end;
7//2 --→q r;
```

This is the same as 7//2→r→q;

The correspondence between a list of text items and POP-2 program is as follows. Syntax words, identifiers and unquoted words are represented by corresponding words in the list. A quoted word is represented by a word with the word quote (consisting of the character quote) before and after it in the list. Integers and reals are represented by integers and reals. String constants are represented by strings.

11.3. Evaluation of program text

A standard function *popval* is provided which will evaluate a list of text items treating it as a POP-2 imperative sequence. The sequence is evaluated immediately. It may contain function definitions and assignments to current variables. Any declarations in it which are not in a function body are global. The list must terminate with the word **goon.** For the correspondence between a list of text items and POP-2 program *see* section 11.2 'Macros'.

The result of the application of *popval* is the result of evaluating the imperative sequence.

popval ∈ *text item list* ⇒ *item*, ... , *item*

Note that *popval* is used to evaluate an imperative sequence at run time and the list of text items may have been produced as the result of computation. It may temporarily affect the standard variable *proglist* (*see* section 9.1 'Input').

Example:

```
1 → a; popval([vars x; a+2 → x; x*x goon]) ⇒
**9
```

The standard routine *setpop* may be applied in the imperative sequence. This restores the system to execute mode. The stack is cleared. The variable currently associated with any identifier is not altered. After *setpop* has been applied the rest of the imperative sequence is ignored and all function bodies currently being evaluated are abandoned. The system then evaluates the next program element. *setpop* may also be applied in a function body.

setpop ∈ () ⇒ ()

ACKNOWLEDGMENTS

This language is a development of R. J. Popplestone's 'POP-1' programming language (see the paper in this volume). The debt to the ALGOL, LISP, CPL and

ISWIM programming languages should be obvious. We are indebted to a number of people in this department and elsewhere for helpful discussion and criticism, to Dr David Park who contributed to discussion of the storage control scheme and to Mrs Margaret Pithie and Miss Eleanor Kerse who typed this report. Mr M. Healy kindly pointed out a number of errors and obscurities in a draft.

The work has been undertaken on a grant from the Science Research Council under the supervision of Dr D. Michie, whose encouragement has been invaluable.

INDICES

TECHNICAL TERMS

SYNTAX DEFINITIONS

STANDARD FUNCTIONS AND VARIABLES

SYNTAX WORDS

INDEX